

تعلم لغة Go بسهولة

كل شيء تحتاج لمعرفته لتبدء البرمجة مع لغة جو

آلان دونوفان

بريان كيرنيغان

تعلم لغة Go بسهولة

كتابك لتعلم واحتراف لغة جو

تأليف:

آلان دونوفان **بريان كيرنيغان**

الإعداد والإشراف والمراجعة للنسخة العربية:

فهد بن عامر السعيدي

ترجمة:

منى هدهد

وسام قواسمه

محمد عبد العزيز

دينا ماهر

عبد اللطيف ايمش

فهد السعيدي

بدعم من:

وادي التقنية

www.itwadi.com

نسخة ١.٠

١٤٣٩هـ - ٢٠١٨م

رخصة الكتاب

أصل هذا العمل ترجمة متصرفة لكتاب The Go Programming Language ، تأليف آلان دونوفان وبريان كيرنيغان، قام بالترجمة العربية موقع وادي التقنية وإشراف عام ومراجعة فهد بن عامر السعيدي.صورة الغلاف مشتقة من عمل إيريك زيليا لغلاف كتاب Go Bootcamp.

إذا صادفت أي من الأخطاء الإملائية واللغوية أو في الترجمة يرجى مراسلتنا عبر هذه الصفحة:

https://itwadi.com/contact

عن المؤلفين

ألان دونوفان، مهندس استشاري في قسم البنية التحتية في شركة جوجل، متخصص في أدوات تطوير البرمجيات. بدأ العمل ضمن فريق لغة جو منذ ٢٠١٢م في تصميم المكتبات و الأدوات للتحليل الساكن. وهو مؤلف العديد من الأدوات منها: `oracle` و `godoc-analysis` و `eg` و `gorename` .

بريان كيرنغان عمل في مركز أبحاث علوم الحاسوب في مختبرات بيل حتى عام ٢٠٠٠م، في مجال لغات يونكس وأدواته. يعمل حاليا بروفيسور في قسم علوم الحاسوب في جامعة Princeton. وهو مؤلف مشارك للعديد من الكتب من ضمنها The C Programming Language و The Practice of Programming.

عن وادي التقنية

وادي التقنية موقع تقني عربي يُعنى بتتبع أخبار البرمجيات الحرة والمواد التعليمية المتعلقة بها، يكتب فيه عدد من المتطوعين المهتمين بالبرمجيات الحرة والتقنية بشكل عام، يهتم وادي التقنية بمواضيع مثل أنظمة التشغيل الحاسوبية و الهاتفية ، لغات البرمجة ، مكتبات البرمجية ، تقنيات الويب ، أخبار شركات البرمجة الكبرى ، المصادر المفتوحة ، العتاد ، و أجهزة الحاسوب والهواتف.

وادي التقنية قام بدعم كتابة العديد من الكتب التقنية في مجال البرمجيات الحرة ومفتوحة المصدر، وتوفيرها مجانا للمستخدم التقني العربي، من أهم الكتب التي دعمها وادي التقنية:

تعلم جافا سكربت، دفتر مدير دبيان، سطر أوامر لينُكس، انطلق في انكسكيب، تعرف على البرمجيات الحرة، بوستجريسكل كتاب الوصفات.

لزيارة وادي التقنية وتقديم الدعم له من هنا:

http://itwadi.com/

مقدمة لا بد لها:

إن سوق لغات البرمجة شهد قدوم العديد من اللغات التي أحدثت تأثيرا كبيرا في صناعة تقنية المعلومات، وأحد هذه اللغات التي أصبحت شعبيتها تنمو بشكل متزايد ومطرد، لغة جو ، ابنة شركة جوجل المدللة.

جاءت لغة جو لتكون لغة عصرية ولكن في نفس قوة لغة السي، ولذا يحلو للبعض أن يطلق عليها "سي العصرية" وبدعم سخي وقوي من شركة جوجل، استطاعت أن تجذب اهتمام الكثير من الشركات والأفراد لتكون اللغة المعتمدة في تقنيات الحوسبة السحابية.

وكالعادة لا تتوفر العديد من المصادر العربية التعليمية لكل التقنيات الحديثة، ولكني آمل أن يكون هذا العمل باكورة أعمال أخرى كما حدث مع كتابتي "تعلم بايثون بسهولة" (للأسف لم يكتمل) عندما أطلقته في عام ٢٠٠٧م حيث لم تكن هناك العديد من الكتب العربية حول لغة بايثون في تلك الفترة، أما الآن ولله الحمد فقد توفرت العديد من المصادر الغنية والمتعمقة في لغة بايثون.

نحن في وادي التقنية نضع هذا الكتاب ولا نطمع في شيء غير نشر العلم، و قد بذلنا مالنا و جهدنا و وقتنا في سبيل ذلك، وقد عزمنا على الأمر لا نألوا على المخذلين من صناع القرار أو إعراض الناس عن القراءة، إنا نؤمن بأن نشر العلم بلسان أهل البلد يفتح فرصة لطامحين لتغيير وتحسين من قدراتهم و حظوظهم من العلم.

وفي الختام ندعو كل من لديه مشروع لتأليف كتاب أو قدرة على الترجمة الجيدة باللغة العربية أن يتواصل معنا، لعل الله أن يوفقنا على إخراج المزيد من الكتب المفيدة في مجال تقنية المعلومات.

نسأل الله القبول و التوفيق.

فهد بن عامر السعيدي

سلطنة عمان

فهرس المحتويات

مقدمة

Go هي لغة برمجة مفتوحة المصدر تجعل تطوير البرامج ذات الكفاءة والموثوقية والبساطة أمرا سهلا" (من موقع لغة جو golang.org)

طورت لغة البرمجة جو في سبتمبر 2007 من قبل روبرت غريز ايمر, ,روب بايك، و كين ثومبس، اللذين كانوا يعملون في جوجل، و أعلن عنها في نوفمبر 2009. وكانت أهداف لغة البرمجة وأدواتها المصاحبة لها أنها تهدف أن تكون معبرة وفعالة في كل من مرحلة الترجمة والتنفيذ، وفعالة في كتابة برامج موثوقة وقوية.

تشبه لغة جو سطحيًا لغة c ، فهي مثل لغة c، حيث أنها أداة يستخدمها المبرمجون المحترفون في تحقيق أقصى قدر من التأثير مع الحد الأدنى من الوسائل، ولكنها أفضل بكثير من النسخة المحدثة من c ، حيث أنها استعارت وكيفت أفكار جيدة من العديد من اللغات الأخرى، بينما تجنبت المميزات التي يمكن أن تجعل الشفرة معقدة وغير موثوقة، فطريقة تعاملها مع التزامن جديدة وفعالة، وأسلوبها مع تجريد البيانات والبرمجة كائنية التوجه مرنة بشكل غير عادى. وهي أيضا تملك إدارة آلية للذاكرة أو ما يسمى مجمع النفايات garbage collection.

لغة جو مناسبة بشكل خاص لبناء البنية التحتية مثل خوادم الشبكات، والأدوات والأنظمة الخاصة بالمبرمجين، ولكنها في الحقيقة لغة متعددة الأغراض وتجد الاستخدام في مجالات متنوعة مثل الرسومات، وتطبيقات الهاتف النقال، والتعلم الآلي. فقد أصبحت شعبية كبديل للغات البرمجة ذات الأنواع غير المحددة، لأنها توازن بين التعبيرية والسلامة: حيث تعمل برامج جو عادة أسرع من البرامج المكتوبة باللغات الديناميكية ، وتعاني أقل من الانهيارات بسبب أخطاء الأنواع غير المتوقعة.

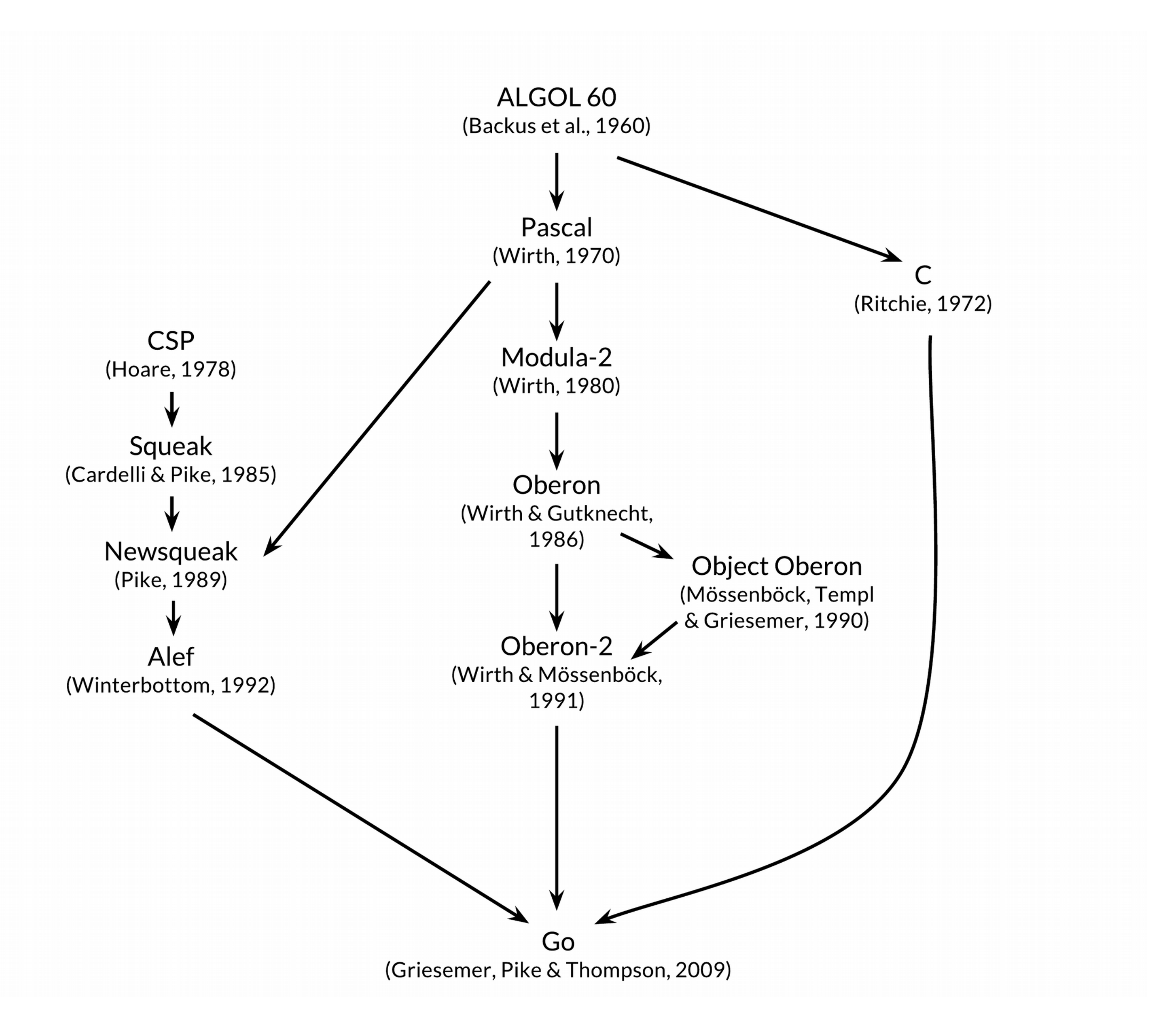
جو هي لغة مفتوحة المصدر، لذا فإن الشفرة المصدرية لمترجمها ، ومكتباتها ، وأدواتها متوفرة مجانًا لأي شخص، وتأتي المساهمات في المشروع من مجتمع عالمي نشط. تعمل لغة جو على أنظمة Unix مثل Linux, FreeBSD, OpenBSD, Mac OS X وعلى Plan9 و Microsoft Windows . البرامج المكتوبة في واحدة من هذه البيئات تعمل عمومًا دون تعديل على المنصات الأخرى.

يهدف هذا الكتاب إلى مساعدتك على البدء في استخدام لغة Go على نحو فعال وعلى الفور، واستخدامها بشكل جيد، والاستفادة الكاملة من ميزات لغة جو والمكتبات القياسية، لكتابة برامج واضحة ،ومتمكنة، وذات كفاءة.

أصول لغة جو

مثل الأنواع البيولوجية، اللغات الناجحة تنجب الأبناء الذين يضمون مزايا أسلافهم؛ يؤدي التهجين أحيانا إلى قوة مدهشة؛ وفي أحيان كثيرة، تظهر ميزة جديدة جذرية من دون سابقة. يمكننا أن نتعلم الكثير من خلال النظر في المؤثرات التي جعلت لغة ما بهذا الشكل، وما البيئة التي كيفت اللغة لها.

يبين الشكل أدناه أهم لغات البرمجة السابقة المؤثرة على تصميم لغة Go.



توصف أحيانا لغة Go بأنها " لغة شبيهة بالسي "، أو سي القرن الحادي والعشرين". ورثت لغة جو من السي القواعد اللغوية، وتعابير التحكم ، وأنواع البيانات الأساسية ، ونمط التمرير بالقيمة، والمؤشرات، وزيادة على ذلك، تركيز لغة السي على البرامج التي تترجم إلى لغة الآلة فعالة، وتتعاون بشكل طبيعي مع جميع أنظمة التشغيل الحالية.

ولكن هناك أجداد آخرون في شجرة عائلة لغة جو. فأحد التيارات التأثير الرئيسية يأتي من لغات Niklaus Wirth ، بدءًا من باسكال . ففكرة الحزم مستوحاة من لغة Modula-2. وقضت لغة Oberon على التمييز بين ملفات واجهة الوحدة البرمجية و ملفات تطبيق الوحدة . وأثرت لغة Oberon-2 في البناء اللغوي للحزم، والاستيراد، والإعلان عن المتغيرات، وزودت Object Oberon بالقواعد اللغوية لجمل الإعلان عن الدوال.

وهناك سلالة أخرى بين أسلاف لغة جو، التي جعلت من لغة جو مميزة من بين لغات البرمجة الحديثة، وهي سلسلة من اللغات البحثية غير المعروفة التي طورت في مختبرات بيل، مستوحاة جميعا من مفهوم تواصل العمليات المتسلسلة (CSP) من بحث توني هوير في عام 1978 في أساسيات التزامن. ففي مفهوم (CSP)، البرنامج هو تكوين متوازي من العمليات التي ليس لها حالة مشتركة؛ وتتواصل العمليات وتتزامن باستخدام القنوات. غير أن برنامج (CSP) الخاص ب هوير كان لغة رسمية لتفسير المفاهيم الأساسية للتزامن، وليس لغة برمجة لكتابة برامج قابلة للتنفيذ.

بدأ العالم روب بايك وآخرون في تجربة تطبيقات CSP كلغات فعلية. كانت أول لغة تدعى سكويك Squeak ('' لغة للتواصل بين الفئران '')، حيث وفرت لغة للتعامل مع أحداث الفأرة و لوحة المفاتيح، مع قنوات أنشئت بشكل ساكن. وأعقبتها لغة نيوسكويك، التي وفرت كلمات و تعابير شبيه بلغة السي وطريقة تحديد الأنواع شبيه بلغة باسكال. وكانت لغة وظيفية بحتة مع مجمع النفايات (garbage collection)، ومرة أخرى هدفت لإدارة أحداث لوحة المفاتيح، والفأرة، والنوافذ وأصبحت القنوات فيها قيما من الدرجة الأولى ، وتنشأ ديناميكيًا وقابلة للتخزين في المتغيرات.

دفع نظام التشغيل plan 9 في تنفيذ هذه الأفكار للأمام في لغة تسمى ألف Alef . حاولت لغة ألف أن تجعل لغة نيوسكويك لغة برمجة لأنظمة التشغيل قابلة للتطبيق. ولكن حذفها لمجمع النفايات garbage collection جعل التزامن عملية صعبة جدا.

تظهر المعماريات الأخرى في لغة جو أثر الجينات الوافدة منتشرة هنا وهناك. فمثلا فكرة تصريح iota جاءت من لغة APL ، و النطاق المعجمي مع الدوال المتفرعة جاءت من لغة Scheme (ومعظم اللغات بعدها). أيضا نجد أنها جاءت بطفرات جديدة. أحد إبداعات لغة Go حيث تقدم مصفوفات ديناميكية ذات وصول عشوائي فعال ولكنها أيضا تسمح ترتيبات التشارك المعقدة للقوائم المرتبطة. كذلك تعبير defer يعتبر جديدا في لغة جو.

مشروع لغة Go

كل لغات البرمجة تعكس فلسفة صانِعِيها والتي غالبة ما تحوي على محتوى كبير من ردة الفعل للأشياء الناقصة في اللغات السابقة. فقد ظهر مشروع لغة Go من خيبة الأمل في العديد من برامج الأنظمة في جوجل التي كانت تعانى من الازدياد المفرط في التعقيد. (هذه المشكلة ليست حصرا على جوجل بأي حال).

وكما صاغها روب بايك:"أن العمليات المعقدة متضاعفة": حيث إنه إذا تطلبت مشكلة ما أن تجعل جزءا واحدا من النظام أكثر تعقيدًا، فإن ذلك سيضيف - بكل تأكيد ولو على المدى البعيد - تعقيدًا إلى الأجزاء الأخرى. ومع الضغط المستمر لإضافة ميزات وخيارات وإعدادات ولنشر الكود بسرعة ، فإنه من السهل غض الطرف عن البساطة، على الرغم من أنها هي المفتاح للبرامج الجيدة على المدى الطويل.

تتطلب البساطة المزيد من العمل في بداية المشروع، لتحويل الفكرة إلى جوهرها، وإلى المزيد من الالتزام المستمر على مدى حياة المشروع للتمييز بين التغييرات الجيدة من السيئة أو الضارة . ومع بذل جهود كافية، يمكن استيعاب تغيير جيد دون أي تنازلات في البساطة و هذا ما دعاه فريد بروكس '' النزاهة المفاهيمية '' للتصميم، ولكن لا يمكن ذلك مع التغيير السيئ، أما التغيير الضار فإنه يضحي بالبساطة من أجل الراحة الخادعة. إن بساطة التصميم هي وحدها التي تجعل نظاما ما مستقرا وآمنا و ومتماسكا أثناء تطوره ونموه.

يشتمل مشروع لغة Go اللغة نفسها ، وأدواتها و مكتباتها البرمجية، وأخيرًا وليس آخرًا، ثقافة البساطة الثورية المحيطة بها. وكلغة حديثة ذات مستوى عالى، فإنها استفادت من المجيء متأخرا، حيث تأسست بشكل جيدا جدا، فهي تملك مجمع النفايات (garbage collection) ، ونظام الحزم(package) ، ودوال ذات التصنيف الأول، ونطاقا معجميا، وواجهة لاستدعاء النظام، والسلاسل النصية غير القابلة للتغيير والتي يرمّز النص فيها غالبا في هيئة UTF-8 .ولكن لديها عدد قليل نسبيا من المميزات ومن غير المرجح أن تضيف المزيد. فعلى سبيل المثال، فهي لا تحتوي على تحويلات الرقمية الضمنية أو توابع بناءة أو توابع مدمرة أو التحميل الزائد للعوامل، أو قيم افتراضية للمعاملات، أو وراثة أو العمومية أو الاستثناءات، أو مايكروات، أو تعريفات الدوال، أو توفير تخزين للخيوط المحلية. إنها لغة ناضجة ومستقرة، وتضمن توافقية مع الإصدارات السابقة: حيث يمكن للبرامج القديمة بلغة Go أن تترجم و تنفذ بواسطة الإصدار الأحدث من المترجم و المكتبات الأساسية.

لغة Go لديها ما يكفي من نظام الأنواع لتجنب معظم الأخطاء التي يقع فيها المبرمجون باللغات الديناميكية، ولكنها أيضا لديها نظام أبسط بالمقارنة مع باقي اللغات ذات الأنواع المحددة. وهذا يؤدى في بعض الأحيان إلى عزل أجزاء برمجية"غير معروفة النوع" داخل إطار واسع من الأنواع. ولا يصل المبرمجون بلغة Go إلى مستوى الذي يصله مبرمجون c++ أو مبرمجون Haskell لتعبير عن خصائص السلامة كما تتطلبه خصائص الأنواع المحددة. ولكن في التطبيق العملي، تعطي لغة جو المبرمجين كل الأمان و سرعة أداء البرنامج الذي تقدمه لغات ذات الأنواع القوية من دون الحاجة إلى التعقيدات المملة المتعلقة بها.

تشجع لغة Go على الوعي بتصميم أنظمة الحواسيب المعاصرة، وخاصة أهمية الموقع. إن كل أنواع بياناتها المضمنة ومعظم هياكل البيانات المكتبية صممت للعمل بشكل طبيعي دون تهيئة صريحة أو توابع بناءة ضمنية، لذلك يوجد عدد قليل نسبيا من حجوزات الذاكرة و الكتابة للذاكرة مخفية في الشفرة. وتستوعب الأنواع الجامعة للغة جو (الهياكل

والمصفوفات) عناصرها مباشرة، وتتطلب تخزينا أقل وحجوزات في الذاكرة أقل ومؤشرات غير مباشرة قليلة مقارنة مع اللغات التي تستخدم حقول غير مباشرة. وبما أن الحاسوب الحديث هو آلة متوازية، فإن جو لديها ميزات التزامن مبنية على أساس CSP ، كما ذكرنا سابقا. فأحجام المكدسات المحلية لخيوط جو أو goroutines صغيرة مبدئيا جدا حتى أن إنشاء وحدة روتين-جو غير مكلف إطلاقا وإنشاء مليون وحدة منها يعتبر أمرا عمليا.

مكتبات لغة جو القياسية ، توصف عادة مثل العبارة "البطاريات مضمنة"، حيث تزودنا بكتل للبناء، ودوال API للتعامل مع الدخل/الخرج، ومعالجة النصوص، والرسومات، والتشفير، والشبكات، والتطبيقات الموزعة، مع دعم العديد من تنسيقات الملفات القياسية والبروتوكولات. تستخدم المكتبات والأدوات الاتفاقيات استخداما واسع النطاق للحد من الحاجة إلى التكوين والشرح، وبالتالي تبسيط منطق البرنامج وجعل برامج جو المتنوعة أكثر مماثلة لبعضها البعض، وبالتالي أسهل للتعلم. تستخدم المشاريع التي أنشئت باستخدام أداة go أسماء الملفات والمعرفات والتعليقات الخاصة في بعض الأحيان لتحديد جميع المكتبات وبرامج التنفيذ والاختبارات والمعايير والأمثلة والمتغيرات الخاصة بالمنصات والوثائق الخاصة بالمشروع. فالشفرة المصدرية لجو تحتوي على مواصفات البناء بداخلها.

محتوى الكتاب

نفترض في هذا الكتاب أنك على علم بإحدى لغات البرمجة الأخرى و قد برمجت بها . سواء عملت بلغة تستخدم المترجم مثل c و ++c و java ، أو بلغة تستخدم المُفسر مثل Python و Ruby و JavaScript ، لذلك نحن لن نشرح كل شيء كما لو كان الوضع للمبتدئ . ستكون بناء الجملة السطحية مألوف، وكذلك المتغيرات والثوابت، والتعبيرات، وتدفق التحكم، والوظائف.

الفصل الأول يشرح البنية الأساسية للغة جو . قُدم من خلال اثني عشر برامج للاستخدامات اليومية مثل القراءة والكتابة إلى الملفات، تهيئة النصوص، وإنشاء الصور، والتواصل مع عملاء الإنترنت والخوادم.

ويصف الفصل الثاني العناصر الأساسية المكونة للغة Go: الإعلانات، والمتغيرات، وأنواع البيانات الجديدة، والحزم والملفات، والنطاق. يناقش الفصل الثالث الأرقام، والعمليات المنطقية ، والسلاسل النصية، والثوابت، ويشرح كيفية معالجة نصوص يونيكود. ويصف الفصل الرابع الأنواع المركبة، وهي الأنواع المبنية من الأنواع الأكثر بساطة باستخدام المصفوفات والخرائط والهياكل والشرائح، طريقة جو للقوائم الديناميكية. ويغطي الفصل الخامس الدوال ويناقش التعامل مع الأخطاء، وجمل panic و recover و defer.

تعتبر الفصول من 1 إلى 5 هي الأساسيات، وهي أمور تشكل جزءا من أي لغة حتمية رئيسية. تختلف صيغة جو وأسلوبها في بعض الأحيان عن لغات أخرى، ولكن معظم المبرمجين سيتكيفون معها بسرعة. وتركز الفصول المتبقية على مواضيع حيث تشرح تفاصيل لغة جو الأقل تقليدية: الدوال، والواجهات، والتزامن، والحزم، والاختبار، والانعكاس.

جو لديها نهج غير عادي للبرمجة الكائنة التوجه. فلا توجد وراثة الصفوف،أو في الواقع أي صفوف، فالكائنات ذات السلوك المعقد تصنع من الكائنات البسيطة بالتركيب ولكن ليس بالوراثة. ويمكن للدوال أن ترتبط بأي نوع مخصص من قبل المستخدم ،وليس فقط بالهياكل، والعلاقة بين أنواع الملموس والأنواع المجردة (واجهات) هو ضمني، لذا يمكن أن يفي نوع ملموس بواجهة لم يكن مصمم النوع على علم بها. يغطي الفصل السادس الدوال بينما يغطى الفصل السابع الواجهات.

ويعرض الفصل الثامن طريقة جو في التزامن، الذي يقوم على فكرة توصيل العمليات المتسلسلة (CSP)، التي تجسدها رويتينات جو goroutines والقنوات channels .

ويوضح الفصل التاسع الجوانب التقليدية للتزامن على أساس المتغيرات المشتركة.

ويصف الفصل العاشر الحزم، وآلية تنظيم المكتبات. ويبين هذا الفصل أيضا كيفية الاستفادة الفعالة من أداة go، والتي تقوم بالترجمة والاختبار، وقياس الأداء، وتنسيق البرنامج، والتوثيق، والعديد من المهام الأخرى، وكل ذلك من خلال أمر واحد.

يتناول الفصل الحادي عشر الاختبارات ، حيث انتهجت جو طريقة مميزة خفيفة الوزن ، وتجنبت أطر التجريد لصالح المكتبات والأدوات البسيطة. وتوفر مكتبات الاختبار أرضية قوية يمكن البناء عليها تجريدات أكثر تعقيدا إذا لزم الأمر.

ويناقش الفصل الثاني عشر الانعكاس، وهي قدرة البرنامج على فحص تمثيله الخاص أثناء التنفيذ. الانعكاس أداة قوية، على الرغم من أنه يجب استخدامها بحذر؛ يشرح هذا الفصل إيجاد التوازن الصحيح من خلال إظهار كيفية استخدامها في تنفيذ بعض المكتبات المهمة.

يشرح الفصل الثالث عشر تفاصيل البرمجة منخفضة المستوى الذي يستخدم حزمة unsafe للتغلب على نظام أنواع جو، وعندما يكون ذلك مناسبا.

كل فصل يحتوي على عدد من التمارين التي يمكنك استخدامها لاختبار فهمك للغة Go، واستكشاف الأفكار الجديدة من أمثلة الكتاب.

جميع شفرات الأمثلة الأكثر سهولة في الكتاب متاحة للتحميل من مستودع جت العام في gopl.io . يعرف كل مثال من خلال مسار استدعاء الحزمة، ويمكن جلبه بسهولة، وبناءه، وتثبيته باستخدام الأمر go get . ستحتاج إلى اختيار دليل ليكون مساحة عمل جو وتعيين متغير البيئة GOPATH ليشير إليه. ستنشئ أداة go الدليل إذا لزم الأمر. فمثلا:

```
$ export GOPATH=$HOME/gobook           # choose workspace directory
$ go get gopl.io/ch1/helloworld        # fetch, build, install
$ $GOPATH/bin/helloworld               # run
Hello, 世界
```

لكى تتمكن من تجربة المثال أنت بحاجة إلى الإصدار 1.5 من Go .

```
$ go version
go version go1.5 linux/amd64
```

إذا لم تكن لديك أداة go أو لديك نسخة قديمة يمكنك تحديثها من هذا الرابط https://golang.org/doc/install

أين أجد معلومات أكثر

أفضل مصدر لمزيد من المعلومات حول لغة Go هو الموقع الرسمي، https://golang.org، الذي يوفر الوصول إلى الوثائق، بما في ذلك مواصفات لغة البرمجة Go ، والحزم القياسية، وما شابه ذلك. هناك أيضا دروس حول كيفية كتابة لغة Go وكيفية الكتابة بها بشكل جيد، ومجموعة واسعة من الموارد على الإنترنت النص والفيديو التي ستكون مكملات قيمة لهذا الكتاب. مدونة Go في blog.golang.org تنشر بعضا من أفضل الكتابات حول لغة Go، من مثل مقالات عن حالة اللغة، والخطط للمستقبل، وتقارير عن المؤتمرات، وتفسيرات متعمقة لمجموعة واسعة من المواضيع ذات الصلة بلغة Go .

أحد الجوانب الأكثر فائدة من الوصول عبر الإنترنت إلى لغة Go (وللأسف محدودية الكتاب الورقي) هو القدرة على تشغيل برامج جو من صفحات الويب التي تصفها. هذه الخدمة موفرة من قبل صفحة ساحة لعب جو في play.golang.org، ويمكن أن تضمن في صفحات أخرى، مثل الصفحة الرئيسية في golang.org أو صفحات الوثائق التي تخدمها أداة godoc. تجعل صفحة اللعب Playground من السهولة تنفيذ تجارب بسيطة للتحقق من فهم بناء الجملة، أو الدوال ، أو حزم المكتبة مع برامج بسيطة، وبطرق متعددة تأخذ هذه الميزة مكان ميزة read-eval-print loop في اللغات الأخرى. وبسبب عناوينها الثابتة، تعتبر مناسبة جدا لمشاركة المقتطفات شفرة جو مع الآخرين للإبلاغ عن الأخطاء أو تقديم اقتراحات.

وفوق مكان اللعب Playground ، بني موقع جولة لغة Go في tour.golang.org الذي هو عبارة عن سلسلة من الدروس التفاعلية القصيرة حول الأفكار الأساسية في لغة جو ومعمارياتها، وهي جولة مرتبة للمرور عبر كل اللغة.

إن القصور الرئيسي في مكان اللعب Playground والجولة Tour هو أنهما يدعمان فقط المكتبات القياسية التي يمكن استيرادها، فالعديد من المميزات – من مثل الشبكات، على سبيل المثال- ممنوعة لأسباب عملية أو أمنية. كما أنها تتطلب الوصول إلى الإنترنت لتجميع وتشغيل كل برنامج. لذلك من أجل تجارب أكثر تفصيل، سيكون عليك تشغيل برامج Go على جهاز الكمبيوتر الخاص بك. لحسن الحظ عملية التحميل واضحة، لذلك لا ينبغي أن يستغرق أكثر من بضع دقائق لجلب توزيعة جو من golang.org والبدء في كتابة وتشغيل برامج جو الخاصة بك.

وبما أن لغة Go هي مشروع مفتوح المصدر، فإنه يمكنك قراءة الشفرة البرمجية لأي نوع أو وظيفة في المكتبة القياسية على الإنترنت في https://golang.org/pkg ؛ وهو نفس الشفرة التي هي جزء من التوزيعة المنزلة. استخدم هذه الحقيقة لمعرفة كيف يعمل شيء ما، أو للإجابة على الأسئلة حول التفاصيل، أو لمجرد رؤية كيف يكتب الخبراء جو بإتقان.

شكر وتقدير

قرأ كل من روب بايك وروس كوكس، الأعضاء الأساسين في فريق Go، الكتاب عدة مرات بعناية كبيرة. إن تعليقاتهم على كل شيء من اختيار الكلمة إلى بنية العامة وتنظيم الكتاب لا تقدر بثمن. وأثناء الإعداد للترجمة اليابانية، ذهب يوشيكي شيباتا أبعد بكثير من الواجب؛ حيث لاحظت عينه المدققة العديد من التناقضات في النص الإنجليزي والأخطاء في التعليمات البرمجية. نحن نقدر جدا المراجعات الشاملة والتعليقات البناءة على الكتاب بأكملها من بريان غويتز، وكوري كوساك، وأرنولد روبنز، وجوش بليتشير شنيدر، وبيتر وينبرجر.

نحن مدينون لسمير أجمانى، وإيتاي بلابان، وديفيد كراوشاو،و بيلي دونوهو، وجوناثان فينبرج، وأندرو جراند، وروبرت غريزيمر، وجون ليندرمان، ومينوكس ما، وبريان ميلز وبالا ناتاراجان، وكوزموس نيكولاو، وبول ستانيفورث، ونايجل تاو، وهوارد تريكي للعديد من الاقتراحات المفيدة. كما نشكر ديفيد برايلسفورد وراف ليفين للحصول على المشورة في الصف الطباعي.

أطلق محررنا جريج دوينش في أديسون-ويسلي شرارة البدء، وظل متساعدا باستمرار منذ ذلك الحين. وكان فريق الإنتاج في أديسون-ويسلي: جون-فولر، ودايانا إيسلي، وجولي ناهل،و تشوتي براسرتسيث، و باربارا وود - مبدعا؛ فلم يأمل المؤلفون أن يحصلوا على دعم أفضل من هذا.

يود آلان دونوفان أن يشكر: سمير أجماني، وكريس ديمتريو، ووالت دروموند، وريد تاتجي من جوجل بتوفير له الوقت للكتابة؛ وستيفن دونوفان، على نصيحته والتشجيع في الوقت المناسب. وقبل كل شيء زوجته ليلى كاظمي، لحماسها المتواصل ودعمها الثابت لهذا المشروع، على الرغم من ساعات طويلة من الانشغالات والتغييبات عن الحياة الأسرية التي نتجت عنها.

بريان كيرنيغان ممتن للغاية للأصدقاء والزملاء على صبرهم وتحملهم كلما تحرك ببطء على طريق التفاهم، وخاصة لزوجته ميج، التي كانت تدعم كتابة الكتاب بلا حدود.

نيويورك

أكتوبر ٢٠١٥

1- دورة تعليمية

يقدم لك هذا الفصل جولة بين المكونات الأساسية للغة Go. نحن نأمل أن نقدم لك معلومات وأمثلة كافية لجعلك تبدأ العمل بطريقة مفيدة في أسرع وقت ممكن. وتهدف الأمثلة المُقدَّمة في هذا الفصل – وفي الحقيقة الكتاب كله – إلى تقديم مهام ستواجهها في العالم الحقيقي بالفعل. سنحاول في هذا الفصل منحك فكرة عن البرامج المتنوعة التي يمكن كتابتها بلغة Go، والتي تتراوح من معالجة الملفات البسيطة وبعض الرسوميات إلى عملاء وخوادم الإنترنت المتزامنين. لن نشرح كل شيء في الفصل الأول بالتأكيد، ولكن دراسة مثل هذه البرامج بلغة جديدة يمكن أن تكون طريقة بدء فعالة.

يميل المرء بشكل طبيعي عند تعلمه للغة جديدة إلى كتابة الشفرة (Code) بطريقة مشابهة للغة التي يعرفها بالفعل. انتبه إلى هذا التحيّز أثناء تعلمك للغة Go وحاول تجنبه. لقد حاولنا توضيح وشرح طريقة الكتابة الجيدة بلغة جو، لذا استخدم الشفرة المُقدَّمة هنا كدليل عند كتابتك لشفرتك الخاصة.

1.1 مرحبًا أيها العالم (Hello, World)

سنبدأ بمثال "hello, world" التقليدي، والذي يظهر في بداية كتاب "لغة البرمجة C"، الذي نُشر عام 1978. إن لغة C هي صاحبة أكبر تأثير مباشر على لغة Go، ويوضح مثال "hello, world" عددا من الأفكار المحورية.

```
gopl.io/ch1/helloworld
package main
import "fmt"
func main() {
    fmt.Println("Hello, 世界")
}
```

إن لغة Go هي لغة تحويلية، حيث تقوم سلسلة أدوات لغة Go بتحويل برنامج المصدر والأشياء المعتمدة عليه إلى تعليمات بلغة الآلة الأصلية الخاصة بالحاسوب. يمكن الوصول إلى هذه الأدوات من خلال أمر واحد هو الأمر "go" والذي يحتوي بدوره على عدد من الأوامر الفرعية. أبسط هذه الأوامر الفرعية هو الأمر "run" والذي يحول شفرة المصدر من

أحد أو عدة ملفات الشفرة التي تنتهي اسمها بـ go. ، ويربطها مع المكتبات، ثم يُشغّل الملف التنفيذي الناتج. (سنستخدم الرمز $ للدلالة على موجّه الأوامر "command prompt" في هذا الكتاب).

```
$ go run helloworld.go
```

ليس من المفاجئ أن هذا يظهر كـ:

```
Hello, 世界
```

تتعامل لغة Go مع اليونيكود (Unicode) بشكل أصيل، حتى يمكنها معالجة النصوص بكل لغات العالم.

لو كان البرنامج أكثر من مجرد تجربة لمرة واحدة، فمن الأفضل أن تقوم بترجمته لمرة واحدة، ثم تحفظ النتيجة المُحوَّلة لاستخدامها لاحقًا. يتم هذا من خلال الأمر "go build":

```
$ go build helloworld.go
```

ينتج هذا ملفا ثنائيا تنفيذيا اسمه "helloworld" يمكن تشغيله في أي وقت دون أي معالجة إضافية:

```
$ ./helloworld
Hello, 世界
```

لقد صنفنا كل مثال مهم لتذكيرك بأنه بإمكانك الحصول على الشفرة من مخزون شفرة المصدر الخاص بالكتاب في gopl.io:

```
gopl.io/chl/helloworld
```

لو شغلت الأمر go get gopl.io/chl/helloworld فإن هذا سيحضر لك شفرة المصدر ويضعها في الدليل المتوافق معها. سنتحدث عن هذا الموضوع أكثر في القسم 2.6 والقسم 10.7.

لنتحدث الآن عن البرنامج نفسه. تُنظَّم شفرة Go في حِزم، وهي تشبه المكتبات أو الوحدات في اللغات الأخرى. تتكون الحزمة من ملف مصدر .go أو أكثر في دليل واحد يحدد ما تقوم به الحزمة. يبدأ كل ملف مصدر بإعلان عن الحزمة package، وفي هذا المثال package main التي توضح الحزمة التي ينتمي إليها الملف، ويليها قائمة بالحزم الأخرى التي يستوردها الملف، ثم إعلان عن البرنامج المُخزَّن في هذا الملف.

تحتوي مكتبة Go القياسية على ما يزيد عن 100 حزمة للمهام الشائعة مثل المُدخلات والمُخرجات، والفرز، والتلاعب بالنص. كمثال، تحتوي حزمة fmt على وظائف لطباعة المُخرج المُهيأ، وإجراء مسح للمُدخل. والوظيفة Println هي

واحدة من وظائف الخرج الأساسية في fmt، فهي تطبع قيمة واحدة أو أكثر، يفصل بينها بمسافات، مع حرف السطر الجديد في النهاية بحيث تظهر كل القيم كمُخرَج من سطر واحد.

إن الحزمة main مميزة، لأنها تُعرِّف البرنامج التنفيذي القائم بذاته وليس المكتبة. إن الوظيفة الرئيسية function main بداخل الحزمة main مميزة أيضًا – ففيها يبدأ تنفيذ البرنامج. إن أي شيء تقوم به الوظيفة الرئيسية يقوم به البرنامج. ستستدعي الوظيفة الرئيسية وظائف في الحزم الأخرى لأداء جزء كبير من العمل بالطبع، مثل وظيفة fmt.Println.

يجب أن نخبر المترجم بالحزم التي يحتاجها ملف المصدر هذا، وهذا هو دور إعلان الاستيراد import الذي يلي إعلان الحزمة. يستخدم برنامج "hello, world" وظيفة واحدة فقط من حزمة أخرى، ولكن معظم البرامج ستستورد المزيد من الحِزم.

يجب أن تستورد الحزم التي تحتاجها بالضبط. ولن يقوم البرنامج بالترجمة لو كان هناك حزم ناقصة لم تستورد، أو لو كان هناك حزم غير ضرورية. يمنع هذا المطلب الصارم تراكم الإشارات إلى الحزم غير المستخدمة مع تطور البرامج.

يجب أن تَتْبَع إعلانات الاستيراد import إعلان الحزمة package. سيتكون البرنامج بعد هذا من إعلانات الوظائف (functions)، والمتغيرات (variables)، والثوابت (constants)، والأنواع (types) (والتي تظهر من خلال الكلمات المفتاحية func, var, const, type)، ولكن ترتيب الإعلانات غير مهم في أغلب الأحوال. إن هذا البرنامج قصير جدًا حيث أنه يعلن عن وظيفة واحدة فقط، والتي تستدعي بدورها وظيفة واحدة أخرى. لن نُظهر إعلانات الحزمة والاستيراد أحيانًا عند تقديم الأمثلة من أجل توفير المساحة، ولكنها موجودة في ملف المصدر، ويجب أن تكون موجودة لترجمة الشفرة.

يتكون إعلان الوظيفة من الكلمة المفتاحية func، واسم الوظيفة، وقائمة المُعامِلات (خالية في حالة الوظيفة الرئيسية main)، وقائمة النتيجة (خالية هنا أيضًا)، وجسم الوظيفة – العبارات التي تُعرِّف ما تقوم به الوظيفة – موضوعين بين قوسين. سنلقي نظرة أقرب على الوظائف في الفصل الخامس.

لا تتطلب لغة Go فاصلات منقوطة في نهاية العبارات أو الإعلانات، إلا عندما يظهر إعلانين أو أكثر في نفس السطر. وفي الواقع، تتحول السطور الجديدة التي تأتي بعد رموز معينة إلى فاصلات منقوطة، وبالتالي مكان وضع السطور الجديدة مهم بالنسبة للتوزيع الصحيح لشفرة Go. كمثال، القوس الافتتاحي } الخاص بالوظيفة يجب أن يكون في نفس السطور الموجود فيه إعلان الوظيفة وليس في سطر بمفرده، وفي الصيغة x + y، سيكون السطر الجديد مسموح به بعد وليس قبل عامل +.

إن لغة جو تأخذ موقفا قويا من تنسيق الشفرة (code formatting). وتقوم أداة gofmt بإعادة كتابة الشفرة بتنسيق قياسي، والأمر الفرعي fmt في الأداة go يطبق أمر gofmt على كل الملفات في الحزمة المحددة، أو الملفات الموجودة

في الدليل الحالي بشكل اعتيادي. إن كل ملفات المصدر المكتوبة بلغة Go في هذا الكتاب طُبق عليها أمر gofmt، ويجب أن تعتاد على فعل نفس الأمر في شفرتك الخاصة. إن الإعلان عن تنسيق قياسي من خلال أمر مُلزم سيوفر الوقت الذي يضيع في الجدل حول التفاهات، والأهم من ذلك أنه يسمح بتحويلات آلية لشفرة المصدر كانت ستصبح غير معقولة لو كان مسموحا بوضع تنسيقات عشوائية.

يمكن تعديل العديد من محررات النصوص لإجراء الأمر gofmt في كل مرة تقوم فيها بحفظ الملف، بحيث تُحفظ شفرة المصدر الخاصة بك بالتنسيق المناسب دائمًا. أحد الأدوات الأخرى المرتبطة بهذا هي أداة goimports، وهي تتحكم في إدخال وحذف إعلانات الاستيراد حسب الحاجة. إنها ليست جزءًا من التوزيع القياسي، ولكن يمكنك الحصول عليها عبر هذا الأمر:

```
$ go get golang.org/x/tools/cmd/goimports
```

إن الطريقة المعتادة بين معظم المستخدمين لتحميل وبناء الحزم وطريقة إجراء اختباراتهم، وإظهار توثيقهم، إلخ هي أداة go، والتي سنلقي نظرة عليها في القسم 10.7.

1.2 مُعطيات سطر الأوامر

تقوم معظم البرامج بمعالجة بعض المُدخلات لإنتاج بعض المُخرجات، وهذا هو تعريف الحوسبة بشكل كبير. ولكن كيف يحصل البرنامج على بيانات المُدخلات التي يعالجها؟ تنتج بعض البرامج بياناتها الخاصة، ولكن المُدخلات عادة ما تأتي من مصدر خارجي: ملف، أو اتصال بشبكة، أو مُخرج برنامج آخر، أو من مستخدم يكتبها على لوحة المفاتيح، أو من معطيات سطر الأوامر، إلخ. ستناقش في الأمثلة القليلة التالية بعض هذه البدائل، وسنبدأ بمعطيات سطر الأوامر.

تقدم حزمة os وظائف وقيم أخرى للتعامل مع نظام التشغيل بطريقة مستقلة عن المنصة. كما أن معطيات سطر الأوامر تكون متاحة لأي برنامج من خلال متغير اسمه Args، وهذا المتغير هو جزء من حزمة os، من ثم، فإن اسمه في أي مكان خارج الحزمة سيكون os.Args.

إن المتغير os.Args هو شريحة slice من السلاسل النصية. والشرائح هي مفهوم أساسي في لغة Go، وسنتحدث بتفصيل أكثر عنها قريبًا، ولكن حاليًا، فكّر في الشريحة كتسلسلات متغيرة الحجم من عناصر المصفوفة، حيث يمكن الوصول للعناصر الفردية باستخدام d[i]، والتسلسل الفرعي المجاور كـ s[m:n]. ويُقدَم عدد العناصر من خلال len(s). كما هو

الحال في معظم لغات البرمجة، فإن كل الفهرسة في Go تستخدم فواصل "نصف مفتوحة" تتضمن أول فهرس ولكن ليس الأخير، لأنه هذا يبسط المنطق. كمثال، الشريحة s[m:n]، حيث 0 < m < n < len(s)، يحتوي على عناصر n-m.

إن العنصر الأول في os.Args، وهو os.Args[0]، هو اسم الأمر نفسه، والعناصر الأخرى هي المُعطيات التي تُقدم للبرنامج عندما يبدأ التنفيذ. إن تعبير الشريحة (Slice Expression) الذي يبدأ بالشكل s[m:n] ينتج عنه شريحة تشير إلى العناصر من m إلى n-1، لذا العناصر التي نحتاجها في مثالنا التالي هي العناصر الموجودة في الشريحة [os.Args[1:len(os.Args)] . لو حذفت m أو n، فإنها سترجع بشكل تلقائي إلى 0 أو len(s) بالترتيب، لذا يمكننا تقديم الشريحة المرغوبة في شكل مختصر كـ [:os.Args[1.

إليك تطبيق لأمر الصدى Unix echo، والذي يطبع معطيات سطر الأوامر في سطر واحد. وهو يستورد حزمتين، تُقدمان كقائمة بين أقواس بدلًا من إعلانات استيراد فردية. إن كلا الشكلين مسموح بهما، ولكن عادة ما يستخدم شكل القائمة. إن ترتيب الاستيرادات لا يهم، حيث تقوم أداة gofmt بفرز أسماء الحزمة بترتيب أبجدي. (عندما يكون هناك نسخ متعددة من مثال واحد، نقوم عادة بترقيمهم حتى تعرف أي مثال نتحدث عنه.)

```
gopl.io/ch1/echo1

// Echo1 يطبع معطيات سطر الأوامر الخاص به
package main
import (
    "fmt"
    "os"
)
func main() {
    var s, sep string
    for i := 1; i < len(os.Args); i++ {
        s += sep + os.Args[i]
        sep = " "
    }
    fmt.Println(s)
}
```

تبدأ التعليقات ب //، وكل النص بداية من // إلى نهاية السطر هو تعليق للمبرمجين، ويتجاهله المترجم. وعن طريق الاتفاق، فإننا نصف كل حزمة في تعليق تالي لإعلان الحزمة مباشرة، وفي الحزمة الرئيسية main يكون هذا التعليق جملة كاملة أو أكثر تصف البرنامج ككل.

إن إعلان var يعلن عن متغيرين هما s و sep، وهما من نوع string . يمكن بدء المتغير كجزء من إعلانه، ولو لم يبدأ صراحة، فإنه يبدأ ضمنًا بقيمة صفرية (Zero value) في النوع الخاص به، وهي 0 للأنواع الرقمية، وتسلسل فارغ " " لسلاسل النصية. من ثم، ففي هذا المثال، يبدأ الإعلان s و sep ضمنًا في تسلسلات فارغة. سنتحدث أكثر عن المتغيرات والإعلانات في الفصل الثاني.

أما بالنسبة للأرقام، تقدم لغة Go المُشغلات الحسابية والمنطقية المعتادة، ولكن عند تطبيقها على السلاسل النصية، يقوم العامل + بعمل وصل للقيم، من ثمّ فإن المعادلة:

```
sep + os.Args[i]
```

تمثل تسلسل للمقاطع sep و os.Args[i]. والجملة المستخدمة في البرنامج:

```
s += sep + os.Args[i]
```

هي جملة تخصيص تُجري تسلسل للقيمة القديمة لـ s مع sep و os.Args[i]، وتعيد تخصيصها لـ s، بحيث تصبح مساوية لـ:

```
s = s + sep + os.Args[i]
```

إن العامل += هو عامل تخصيص. وكل عامل حسابي أو منطقي مثل + أو * له عامل تخصيص مناظر.

إن برنامج الصدى كان يمكن أن يطبق مخرجه في حلقة متكررة كقطعة واحدة في المرة الواحدة، ولكن هذه النسخة تبني تسلسل بدلًا من ذلك من خلال إدخال نص جديد في النهاية بشكل متكرر.

يبدأ التسلسل s الحياة فارغًا، بمعنى أنه يبدأ بالقيمة " "، وكل لفة عبر الحلقة تضيف بعض النص إليه، وبعد التكرار الأول، يتم إدخال مسافة أيضًا بحيث عندما تنتهي الحلقة، يكون هناك مسافة بين كل مُعطى من المعطيات. إن هذه عملية رباعية قد تكون مُكلفة لو كان عدد المعطيات كبير، ولكن هذا أمر غير مرجح في الصدى. سنعرض عدد من نسخ الصدى المُحسّنة في هذا الفصل والفصل التالي، وتتعامل هذه النسخ مع أي عدم كفاءة حقيقي.

إن متغير مؤشر الحلقة i يُعلَن في الجزء الأول من حلقة for. ورمز := هو جزء من "إعلان المتغير القصير"، وهي جملة تعلن عن متغير أو أكثر، وتحدد لهم الأنواع المناسبة بناء على قيم التمهيدية، وسنتحدث بالتفصيل أكثر عن هذا في الفصل القادم.

إن عبارة الإضافة ++i تضيف 1 إلى i، وهي مكافِئة لـ i+=1، والتي تكافئ بدورها i = i + 1. وهناك عبارة طرح مناظرة هي --i تقوم بطرح 1. هذه عبارات وليست تعبيرات (صيغ) كما هو الحال في معظم اللغات في عائلة C، بالتالي j = i++ غير قانونية، ومجرد إضافة لاحقة فقط، بالتالي تعتبر --i غير قانونية أيضًا.

إن حلقة for هي الحلقة الوحيدة في لغة Go. وهي تحتوي على عدد من الأشكال، أحدها موضح هنا:

```
for initialization; condition; post {
// عبارات صفرية أو أكثر
}
```

لا تُستخدم الأقواس حول المكونات الثلاثة لحلقة for، ولكن الحواصر (braces) إجبارية، وحاصرة الفتح يجب أن تكون في نفس سطر عبارة الرسالة (post statement).

إن عبارة التمهيد (initialization statement) الاختيارية تُنفذ قبل بدء الحلقة، ولو كانت موجودة، يجب أن تكون عبارة بسيطة (simple statement)، وهي إعلان قصير عن متغير، أو عبارة متعلقة بزيادة أو تخصيص، أو مهمة وظيفة. إن الشرط "Condition" هو تعبير منطقي يُقيَّم في بداية كل تكرار للحلقة، ولو أعطي قيمة موجبة، فإن العبارات التي تتحكم فيها الحلقة ستنفذ. تُنفذ عبارة post بعد متن الحلقة، ثم يقيم الشرط مرة أخرى. تنتهي الحلقة عندما يصبح الشرط خاطئًا.

يمكن حذف أي جزء من هذه الأجزاء، ولو لم يكن هناك تمهيد initialization ولا رسالة post، فإن الفاصلات المنقوطة يمكن حذفها كذلك:

```
// حلقة "while" تقليدية
for condition {
// ...
}
```

ولو حُذف الشرط بالكامل في أي من هذه الأشكال، كمثال في:

```
// حلقة تكرار لا نهائية تقليدية
for {
// ...
}
```

فإن الحلقة تصبح لا نهائية، بالرغم من أنه يمكن إنهاءها بطرق أخرى ، مثل عبارة break أو return.

وهناك شكل آخر من حلقات for حيث يمر على نطاق (range) من القيم مثل سلاسل نصية أو شريحة. وللتوضيح، إليك نسخة أخرى من برنامج الصدى echo:

```
gopl.io/ch1/echo2
// Echo2 يطبع معطيات سطر الأوامر الخاص به
package main
import (
    "fmt"
    "os"
)
func main() {
    s, sep := "", ""
    for _, arg := range os.Args[1:] {
        s += sep + arg
        sep = " "
    }
    fmt.Println(s)
```

```
}
```

في كل تكرار للحلقة، ينتج النطاق range زوجا من القيم: الفهرس وقيمة العنصر في هذا الفهرس. نحن لا نحتاج إلى الفهرس في هذا المثال، ولكن صياغة حلقة range تتطلب أننا لو كنا سنتعامل مع عنصر، فيجب أن نتعامل مع الفهرس أيضًا. إن أحد الأفكار ستكون تخصيص الفهرس إلى متغير مؤقت واضح مثل temp وتجاهل قيمته، ولكن لغة Go لا تسمح بالمتغيرات المحلية غير المستخدمة، بالتالي سينتج عن هذا خطأ في الترجمة.

إن الحل هو استخدام مُعرِّف فارغ (blank identifier) اسمه _ (أي شرطة سفلية). يمكن استخدام المُعرف الفارغ كلما احتاجت الصياغة إلى اسم متغير، ولكن منطق البرنامج لم يحتجها، كمثال، للتخلص من فهرس حلقة غير مرغوب فيه عندما نحتاج إلى قيمة العنصر فقط. سيستخدم معظم مبرمجي Go النطاق range و _ على الأرجح لكتابة برنامج الصدى بالطريقة الموضحة أعلاه. ونظرًا لكون فهرسة os.Args ضمنية وليست صريحة، فبالتالي سيكون من الأسهل فعل هذا بشكل صحيح.

تستخدم هذه النسخة من البرنامج إعلان متغير قصير للإعلان عن وتمهيد s و sep، ولكن يمكننا أن نعلن بوضوح أيضًا عن المتغيرات بشكل منفصل. هناك العديد من الطرق للإعلان عن متغير تسلسل، وكلها متكافئة:

```
s := ""
var s string
var s = ""
var s string = ""
```

لماذا تفضّل شكلًا على الآخر؟ الشكل الأول وهو إعلان متغير قصير وهو أكثر الأشكال انضغاطا، ولكن يمكن استخدامه فقط داخل الوظيفة، وليس مع المتغيرات على مستوى الحزمة. يعتمد الشكل الثاني على التمهيد الاعتيادي لقيمة الصفر الخاصة بالتسلسلات، وهي " ". نادرًا ما يُستخدم الشكل الثالث إلا عند الإعلان عن متغيرات متعددة. إن الشكل الرابع صريح بشأن نوع المتغير، ويكون زائدا عن الحاجة عندما يكون مطابق للقيمة المبدئية، ولكنه يكون ضروريا في الحالات الأخرى عندما لا تكون المتغيرات من نفس النوع. وفي الواقع العملي، يجب أن تستخدم نوعا من النوعين الأوليين، مع تمهيد صريح لقول أن القيمة المبدئية مهمة، وتمهيد ضمني لقول أن القيمة المبدئية غير مهمة.

كما ذكرنا أعلاه، تنضم محتويات جديدة تمامًا للتسلسل مع كل تكرار للحلقة. وتصنع عبارة =+ تسلسل جديد من خلال وصل التسلسل القديم، ثم مسافة، ثم معطى جديد، ثم تخصيص تسلسل جديدة لـ s. إن المحتويات القديمة لـ s تُصبح غير مستخدمة بعدها، وبالتالي ستحذف بعد ذلك.

لو كانت كمية البيانات المتضمنة كبيرة، فإن هذا قد يكون مكلفًا. وسيكون الخيار الأكثر بساطة وكفاءة هو استخدام "الوظيفة Join" من حزمة سلاسل النصية strings:

```
func main() {
    fmt.Println(strings.Join(os.Args[1:], " "))
}
```

أخيرًا، لو لم تكن مهتما بشأن التنسيق، ولكنك تريد رؤية القيم، لأجل البحث عن الأخطاء ربما، يمكننا استخدام الأمر Println لتهيئة النتائج لأجلنا:

```
fmt.Println(os.Args[1:])
```

إن نتيجة هذه العبارة تشبه ما سنحصل عليه من string.Join ولكن مع أقواس هلالية محيطة بها. ويمكن طباعة أي شريحة بهذه الطريقة.

تمرين 1.1: عدل برنامج الصدى لطباعة os.Args[0] أيضًا، واسم الأمر الذي يقوم بتشغيله.

تمرين 1.2: عدل برنامج الصدى لطباعة الفهرس وقيمة كل من معاملاته، كل معامل في كل سطر.

تمرين 1.3: قم بتجربة قياس الفارق في وقت التشغيل بين برنامجنا غير الكفء بشكل محتمل والبرنامج الذي يستخدم strings.Join. (يوضح القسم 1.6 جزء من الحزمة الزمنية time، بينما يوضح القسم 11.4 كيفية كتابة اختبارات مقارنة لتقييم الأداء المنهجي.)

1.3 إيجاد السطور المتكررة

إن كل برامج النسخ والطباعة والبحث والفرز والحساب الملفات وغيرها ذات هيكل متشابه: حلقة تكرار للمدخلات، وبعض الحسابات حول كل عنصر، وإنتاج المخرجات أثناء العمل أو في النهاية. سنظهر ثلاث تنويعات مختلفة على البرنامج يسمى dup، وهي مستلهمة جزئيًا من أمر Unix uniq، والذي يبحث عن السطور المتكرر المتجاورة. إن الهياكل والحزم المستخدمة هي نماذج يمكن تعديلها بسهولة.

تطبع النسخة الأولى من dup كل سطر يظهر أكثر من مرة واحدة في المُدخل القياسي، ويُكتب رقمه قبله. يقدم هذا البرنامج عبارة if، ونوع البيانات الخريطة map، وحزمة bufio.

```
gopl.io/ch1/dup1
// Dup1 prints the text of each line that appears more than
// once in the standard input, preceded by its count.
package main
import (
    "bufio"
    "fmt"
    "os"
)
```

```go
func main() {
    counts := make(map[string]int)
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

كما هو الحال مع for، فإن الأقواس لا تُستخدم أبدًا حول الشرط في عبارة if، ولكن الحواصر مطلوبة في الجسم. يمكن أن يكون هناك جزء آخر اختياري يُنفذ لو كان الشرط خاطئ.

تحمل الخريطة map مجموعة من ثنائيات المفتاح/القيمة، وتقدم عمليات زمنية ثابتة لتخزين واستعادة أو اختبار بند في المجموعة. إن المفتاح قد يكون أي نوع يمكن مقارنة قيمته مع ==، إن السلاسل النصية هي المثال الأكثر شيوعًا، وقد تكون القيمة من أي نوع على الإطلاق. إن المفاتيح في هذا المثال هي سلاسل نصية والقيم هي int. إن الوظيفة المدمجة make يمكن أن تصنع خريطة فارغة جديدة، ولها استخدامات أخرى أيضًا. سنناقش الخرائط بشكل مُطوّل في القسم 4.3.

يقرأ dup في كل مرة سطرا واحدا من المُدخلات، و يُستخدم السطر كمفتاح للخريطة، والقيمة المناظرة تتزايد. إن العبارة ++counts[input.Text()] مكافئة للعبارتين التاليتين:

```go
line := input.Text()
counts[line] = counts[line] + 1
```

لا توجد مشكلة لو كانت الخريطة لا تحتوي على المفتاح بعد، وعند رؤية السطر الجديد لأول مرة، يقوم التعبير counts[line] على الجانب الأيمن بتقييم القيمة الصفرية من حيث نوعها، والذي يكون 0 لـ int.

نستخدم حلقة for أخرى قائمة على النطاق لطباعة النتائج، ولكن هذه المرة مع خريطة الحسابات. وكما هو الحال من قبل، فإن كل تكرار ينتج عنه نتيجتين، وهما المفتاح وقيمة عنصر الخريطة الخاصة بهذا المفتاح. إن ترتيب تكرار الخريطة غير محدد، ولكنه يصبح عشوائيًا في الواقع العملي، ويتفاوت من تشغيل إلى آخر. هذا التصميم متعمد لأنه يمنع البرامج من الاعتماد على أي ترتيب محدد تكون نتائجه مضمونة.

ننتقل بعد ذلك إلى حزمة bufio، والتي تساعد على جعل التعامل مع المدخلات والمخرجات يكون ذي كفاءة وسهولة. وواحدة من أكثر خصائصها فائدة هي نوع اسمه Scanner الذي يقرأ المُدخل، ويقسمه إلى سطور أو كلمات، وعادة ما يكون أسهل طريقة لمعالجة المدخل الذي يأتي في سطور في صورته الطبيعية.

يستخدم البرنامج إعلان متغير قصير لإنشاء متغير جديد input يشير إلى bufio.Scanner:

```
input := bufio.NewScanner(os.Stdin)
```

يقرأ الماسح من مُدخل البرنامج القياسي، وكل استدعاء لـ input.Scan() يقرأ السطر التالي ويحذف حروف السطر الجديد من النهاية، ويمكن استعادة النتيجة من خلال استدعاء input.Text(). تكون نتائج وظيفة Scan صحيحة لو كان هناك سطرا، وخاطئة لو لم يكن هناك المزيد من المدخلات.

إن الوظيفة fmt.Printf، مثلها مثل printf في C واللغات الأخرى، تنتج مخرجا مهيأ من قائمة التعبيرات. إن المعطى الأول فيه هو سلسلة نصية منسقة تحدد كيف يجب تهيئة أو تنسيق المعطيات التالية. إن تنسيق كل معطى يتقرر من خلال حرف التحويل، وهو الحرف التالي لعلامة النسبة. كمثال، تُشكّل d% معامل رقم صحيح باستخدام رمز علامة عشرية، بينما s% تتوسع لتشمل قيمة السلسلة النصية.

يمتلك Printf أكثر من دستة من هذه التحويلات، وهو ما يُطلق عليه مبرمجي Go "الأفعال/Verbs". إن هذا الجدول أبعد ما يكون عن توصيف كامل، ولكنه يوضح العديد من الخصائص المتاحة:

d%	رقم صحيح عشري
x, %o, %b%	رقم صحيح في بنظام الست العشرية، والثماني عشرية، والثنائي.
f, %g, %e%	رقم النقطة العائمة: 3.141593e+00 3.141592653589793 3.141593
t%	منطقي: true أو false
c%	rune (نقطة شفرة يونيكود)
s%	سلسلة نصية
q%	تسلسل مقتبس "abc" أو نقطة يونيكود 'c'
v%	أي قيمة قي تنسيق طبيعي
T%	نوع أي قيمة
%%	علامة النسبة المئوية الحرفية (لا معامل)

يحتوي تسلسل التنسيق في dup1 أيضًا على تاب t\ وسطر جديد n\. قد تحتوي حروف السلسلة النصية على "تسلسلات هروب (escape sequences)" لتمثيل الحروف التي لا تكون مرئية بخلاف هذا. لا يكتب Printf سطر جديد بشكل افتراضي. ولكن من خلال الاتفاق، تقوم وظائف التنسيق التي تستخدم أسماء تنتهي بحرف f مثل log.Printf و

fmt.Errorf باستخدام قواعد تنسيق fmt.Print، بينما أن التي تنتهي أسمائها بـ ln تتبع قواعد Println، وتنسق معاملاتها كما لو أنها تنسقها باستخدام v%، ويتبعها سطر جديد.

تقرأ العديد من البرامج إما من مدخلاتها القياسية، كما هو الحال أعلاه، أو من خلال تسلسل من الملفات المُسماة بأسماء معينة. إن النسخة التالية من dup يمكن أن تقرأ معطياتها من المدخل القياسي أو من قائمة بأسماء الملف، باستخدام os.Open لفتح كل واحد منهم:

```
gopl.io/ch1/dup2
// Dup2 prints the count and text of lines that appear more than once
// in the input.  It reads from stdin or from a list of named files.
package main
import (
    "bufio"
    "fmt"
    "os"
)
func main() {
    counts := make(map[string]int)
    files := os.Args[1:]
    if len(files) == 0 {
        countLines(os.Stdin, counts)
    } else {
        for _, arg := range files {
            f, err := os.Open(arg)
            if err != nil {
                fmt.Fprintf(os.Stderr, "dup2: %v\n", err)
                continue
            }
            countLines(f, counts)
            f.Close()
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
func countLines(f *os.File, counts map[string]int) {
    input := bufio.NewScanner(f)
    for input.Scan() {
        counts[input.Text()]++
    }
    // NOTE: ignoring potential errors from input.Err()
}
```

إن الوظيفة os.Open تقدم قيمتين، الأولى هي ملف مفتوح (*os.File) يُستخدم في القراءات التالية بواسطة الماسح Scanner.

والنتيجة الثانية لـ os.Open هي قيمة نوع الخطأ المُدمج error. لو كان err مساوي للقيمة الفارغة المدمجة nil، فإن الملف يكون قج فُتح بنجاح. يُقرأ الملف، وعند الوصول لنهاية المُدخل، يقوم الأمر Close بإغلاق الملف وتحرير أي موارد. من ناحية أخرى، لو كان err ليس nil سيكون هناك شيء ما خاطئ. وفي تلك الحالة، ستصف قيمة الخطأ المشكلة. إن معالجتنا البسيطة للخطأ تطبع رسالة على خرج الخطأ القياسي باستخدام Fprintf و الفِعل (verb) %v، والذي يُظهر قيمة أي نوع في التنسيق الاعتيادي، ثم يقوم dup بمتابعة العمل على الملف التالي، وتدخل عبارة الاستمرار continue في التكرار التالي في حلقة for المغلقة.

إن أمثلتنا السابقة محدودة قليلًا في التعامل مع الأخطاء وهذا في محاولة منا لإبقاء عينات الشفرة بحجم معقول. ومن الواضح أننا يجب أن نفحص الخطأ من os.Open، ولكننا نتجاهل الاحتمالية الأقل وهي أن الخطأ يمكن أن يحدث أثناء قراءة الملف باستخدام input.Scan. سنعلم الأماكن التي تجاهلنا فيها التحقق من وجود أخطاء، وسنتحدث بالتفصيل عن التعامل مع الأخطاء في القسم 5.4.

لاحظ أن استدعاء countLines يسبق إعلانه. وأن الوظائف والهيئات الأخرى على مستوى الحزمة يمكن إعلانها بأي ترتيب.

إن الخريطة هي مرجع "reference" لهيكل البيانات صنع باستخدام make. عند تمرير الخريطة إلى وظيفة، تتلقى الوظيفة نسخة من المرجع، وبالتالي فإن أي تغيير تقوم به الوظيفة في هيكل البيانات الضمني سيظهر عبر مرجع خريطة الطالِب أيضًا. وفي مثالنا، ترى الوظيفة الرئيسية main القيم التي تُدخل خريطة الحسابات بواسطة countLines.

إن نسخ dup أعلاه تعمل في وضع "التدفق" والذي يُقرأ فيه المُدخل ويُقسم إلى سطور حسب الحاجة، لذا من حيث المبدأ، يمكن لهذه البرامج التعامل مع أي مقدار عشوائي من المُدخلات. إن الطريقة البديلة هي قراءة المُدخل بأكمله وإدخاله للذاكرة مرة واحدة، وتقسيمه إلى سطور كله مرة واحدة، ثم معالجة السطور. إن النسخة التالية، dup3، تعمل بتلك الطريقة أيضًا. فهي تقدم وظيفة ReadFile (من حزمة io/ioutil)، والتي تقرأ المحتويات الكاملة للملف المُسمي، وتطبق strings.Split، الذي يقسم السلاسل النصية إلى شريحة من التسلسلات الفرعية. (إن التقسيم Split هو نقيض strings.Join الذي رأيناه سابقًا).

لقد بسّطنا dup3 إلى حد ما. أولا، حيث يقوم بقراءة أسماء الملفات فقط، وليس المدخل القياسي، حيث أن ReadFile يتطلب معطى هو اسم الملف. وثانيًا، نقلنا حساب السطور إلى main مرة أخرى حيث أنه مطلوب الآن في مكان واحد فقط.

```
gopl.io/ch1/dup3 package main
// Dup3 prints the count and text of lines that
// appear more than once in the named input files.
```

```go
package main
import (
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)
func main() {
    counts := make(map[string]int)
    for _, filename := range os.Args[1:] {
        data, err := ioutil.ReadFile(filename)
        if err != nil {
            fmt.Fprintf(os.Stderr, "dup3: %v\n", err)
            continue
        }
        for _, line := range strings.Split(string(data), "\n") {
            counts[line]++
        }
    }
    for line, n := range counts {
        if n > 1 {
            fmt.Printf("%d\t%s\n", n, line)
        }
    }
}
```

يُخرِج ReadFile شريحة بايت (bite slice) يجب تحويلها إلى string حتى يمكن تقسيمها بواسطة strings.Split.

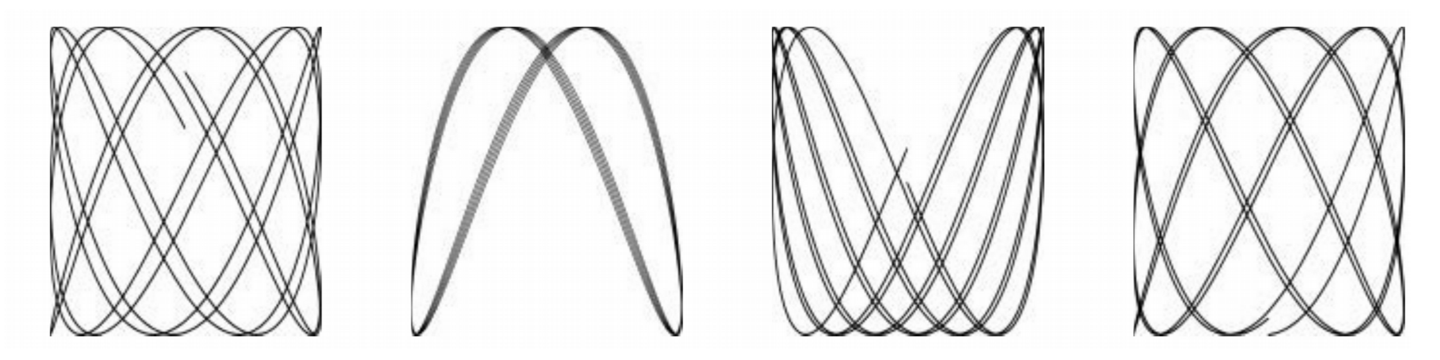
وسنناقش السلاسل النصية وشرائح البايت بشكل مطول في القسم 3.5.4.

تستخدم bufio.Scanner و ioutil.ReadFile و ioutil.WriteFile بشكل غير ظاهر أساليب Read و Write في os.File* ولكن من النادر أن يحتاج معظم المبرمجين للدخول لتلك الروتينات منخفضة المستوى مباشرة. إن الوظائف الأعلى مستوى مثل تلك الخاصة بـ bufio و io/ioutil سهلة الاستخدام أكثر.

تمرين 1.4: عدّل dup2 لطباعة أسماء كل الملفات التي يحدث فيها تكرار للسطور.

1.4 صور GIFs المتحركة (Animated GIFs)

يوضح البرنامج التالي الاستخدام الأساسي لحزم الصورة القياسية image في Go، والتي سنستخدمها لصنع سلسلة من الصور ذات الخريطة المرسومة باستخدام بِت (bit)، ثم ترميز هذه السلسلة كصورة GIF متحركة. إن الصور التي يُطلق عليها أشكال Lissajous وهي عبارة عن تأثيرات بصرية أساسية في أفلام الخيال العلمي في الستينيات. وهي منحنيات بارامترية ناتجة عن الذبذبة التناغمية في بُعدين، مثل موجتيّ sine يُغذيان المُدخلات x و y في راسم الذبذبات. يوضح الشكل 1.1 بعض الأمثلة على هذه الأشكال.



الشكل 1.1: أربع أشكال Lissajous

هناك العديد من البنيات الجديدة في هذه الشفرة، بما في ذلك إعلانات const، وأنواع struct، والقيم الحرفية المُركبة. وعلى العكس من معظم أمثلتنا، تضمّن هذا المثال حسابات النقطة العائمة. وسنناقش هذه المواضيع بشكل موجز فقط هنا، وسنترك الحديث عن معظم هذه التفاصيل إلى الفصول التالية، حيث أن الهدف الأساسي الآن هو إعطاءك فكرة عن كيف تبدو لغة Go، وأنواع الأشياء التي يمكن القيام بها بسهولة باستخدام اللغة ومكتباتها.

```
gopl.io/ch1/lissajous
// Lissajous generates GIF animations of random Lissajous figures.
package main
import (
    "image"
    "image/color"
    "image/gif"
    "io"
    "math"
    "math/rand"
    "os"
)
var palette = []color.Color{color.White, color.Black}
const (
    whiteIndex = 0 // first color in palette
    blackIndex = 1 // next color in palette
)
func main() {
   lissajous(os.Stdout)
}
func lissajous(out io.Writer) {
    const (
        cycles  = 5     // number of complete x oscillator revolutions
        res     = 0.001 // angular resolution
        size    = 100   // image canvas covers [-size..+size]
        nframes = 64    // number of animation frames
        delay   = 8     // delay between frames in 10ms units
    )
    freq := rand.Float64() * 3.0 // relative frequency of y oscillator
    anim := gif.GIF{LoopCount: nframes}
    phase := 0.0 // phase difference
    for i := 0; i < nframes; i++ {
        rect := image.Rect(0, 0, 2*size+1, 2*size+1)
        img := image.NewPaletted(rect, palette)
```

```
        for t := 0.0; t < cycles*2*math.Pi; t += res {
            x := math.Sin(t)
            y := math.Sin(t*freq + phase)
            img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
                blackIndex)
        }
        phase += 0.1
        anim.Delay = append(anim.Delay, delay)
        anim.Image = append(anim.Image, img)
    }
    gif.EncodeAll(out, &anim) // NOTE: ignoring encoding errors
}
```

بعد استيراد الحزمة التي يحتوي مسارها على مكونات متعددة، مثل image/color، سنرجع إلى الحزمة التي يأتي اسمها من المُكون الأخير. من ثم، فإن المتغير color.White ينتمي إلى حزمة image/color، بينما ينتمي gif.GIF إلى image/gif .

إن إعلان const (انظر 3.6) يمنح أسماء للثوابت، وهي القيم الثابتة في وقت التجميع، مثل المعاملات الرقمية الخاصة بالدورات والإطارات والتأجيل. إن إعلانات const – مثلها مثل إعلانات var – يمكن أن تظهر على مستوى الحزمة (وبالتالي تكون الأسماء ظاهرة في أنحاء الحزمة) أو داخل الوظيفة (بحيث تكون الأسماء ظاهرة فقط داخل الوظيفة). إن قيمة الثابت يجب أن تكون رقم أو سلسلة نصية أو قيمة منطقية (Boolean).

إن التعبيرات {...}color.Color [ ] و {...}gif.GIF هي قيم حرفية مركبة (انظر 4.2, انظر 4.4.1)، وهي تدوين مضغوط يمثل أي من أنواع Go المركبة من تسلسل من قيم العنصر. وفي هذا المثال، الأول هو شريحة slice والثاني هو struct.

إن النوع gif.GIF هو نوع بنية struct انظر (انظر 4.4). والبنية هي مجموعة من القيم يُطلق عليها حقول (fields)، وعادة ما تكون ذات أنواع مختلفة، تجمع معًا في جسم واحد يمكن معاملته كوحدة واحدة. إن المتغير anim هو بنية من النوع gif.GIF. تقوم حرفية البنية بإنشاء قيمة البنية يكون حقل Loop-Count الخاص بها مضبوط على nframes، وكل الحقول الأخرى تعطى قيمة صفرية بالنسبة لأنواعهم. إن الحقول الفردية الخاصة بـ struct يمكن الوصول إليها باستخدام علامة dot، كما هو الحال في آخر مهمتين حيث حدثت حقول anim الخاصة بالتأجيل Delay والصورة Image فيهما بشكل صريح.

إن وظيفة lissajous بها حلقتين متداخلتين، تقوم الحلقة الخارجية ب 64 تكرار، كل منها ينتج إطار تحريك واحد، وهي تصنع صورة جديدة 201×201 بلوحة ألوان مكونة من لونين، أبيض وأسود. إن كل البيكسلات (pixels) توضع في البداية على لوحة ألوان ذات قيمة صفرية (اللون الصفري في لوحة الألوان)، والتي نضبطها على اللون الأبيض. وكل مرور عبر الحلقة الداخلية ينتج عنه صورة جديدة من خلال ضبط بعض البيكسلات على اللون الأسود. تُرفق النتيجة باستخدام وظيفة الإرفاق append المدمجة (انظر 4.2.1) على قائمة إطارات في anim، بجانب تأجيل محدد 80 مث.

أخيرًا، نقوم بترميز تسلسل الإطارات والتأجيلات في هيئة GIF، ويُكتب على التدفق الخارجي للمُخرَج. إن نوع المُخرج هو io.Writer، والذي يسمح لنا أن نكتب على نطاق واسع من الوجهات المحتملة كما سنوضح قريبًا.

تشغّل الحلقة الداخلية ذبذبتين، وهما ذبذبة x وهي وظيفة sine وحسب، وذبذبة y وهي أيضًا sinusoid، ولكن ترددها بالنسبة لذبذبة x هي رقم عشوائي بين صفر و 3، ومرحلتها بالنسبة للذبذبة x تكون صفر في البداية ولكنها تزيد مع كل إطار تحريك. تعمل الحلقة حتى يكمل التذبذب x خمس دورات كاملة، وفي كل خطوة، يستدعي SetColorIndex لتلوين البيكسلات المناظرة لـ (x, y) باللون الأسود، والذي يوجد في الموضع 1 في لوحة الألوان.

تستدعي الوظيفة الأساسية وظيفة lissajous، وتوجهها إلى كتابة المُخرج القياسية، بحيث ينتج هذا الأمر GIF متحركة بإطارات كتلك الموجودة في الشكل 1.1:

```
$ go build gopl.io/ch1/lissajous
$ ./lissajous >out.gif
```

تمرين 1.5: قم بتغيير لوحة ألوان برنامج lissajous للون الأخضر على الأسود لإضافة مزيد من الأصالة. ولصنع لون الويب #RRGGBB استخدِم color.RGBA{0xRR, 0xGG, 0xBB, 0xff}، حيث كل زوج من الأرقام الستة العشرية يمثل شدة عنصر اللون الأحمر أو الأخضر أو الأزرق للبيكسل.

تمرين 1.6: عدل برنامج Lissajous بحيث ينتج صور بألوان متعددة من خلال إضافة المزيد من القيمة للوحة الألوان، ثم اعرضها من خلال تغيير المعطى الثالث في Set-ColorIndex بطريقة ما.

1.5 جلب عنوان (URL)

إن الوصول للمعلومات من الإنترنت مهم مثله مثل أهمية الوصول لنظام الملفات المحلية في العديد من التطبيقات. تقدم لغة Go مجموعة من الحزم المُجمّعة تحت حزمة net، مما يجعل من السهل إرسال واستقبال المعلومات عبر الإنترنت، وبناء اتصال شبكي منخفض المستوى، وإعداد الخوادم، التي تكون فيها خصائص Go المتزامنة (التي سنقدمها في الفصل الثامن) مفيدة بشكل خاص.

لتوضيح صورة استعادة المعلومات عبر HTTP، سنقدم برنامج بسيط اسمه "fetch" يقوم بجلب محتوى كل رابط محدد ويطبعه كنص غير مفسر، وهو مُستلهم من الأداة curl التي لا تُقدَر بثمن. من الواضح أنه بإمكاننا فعل المزيد من الأمور باستخدام هذه البيانات، ولكن هذا يوضح الفكرة الأساسية. وسنستخدم هذا البرنامج بشكل متكرر في هذا الكتاب.

```
gopl.io/ch1/fetch
```

```
// Fetch prints the content found at each specified URL.
package main
import (
	"fmt"
	"io/ioutil"
	"net/http"
	"os"
)
func main() {
	for _, url := range os.Args[1:] {
		resp, err := http.Get(url)
		if err != nil {
			fmt.Fprintf(os.Stderr, "fetch: %v\n", err)
			os.Exit(1)
		}
		b, err := ioutil.ReadAll(resp.Body)
		resp.Body.Close()
		if err != nil {
			fmt.Fprintf(os.Stderr, "fetch: reading %s: %v\n", url, err)
			os.Exit(1)
		}
		fmt.Printf("%s", b)
	}
}
```

يقدم هذا البرنامج وظائف من حزمتين، هما net/http و io/ioutil. إن وظيفة httt.Get تقدم طلب HTTP، ولو لم يكن هناك خطأ، فإنها ستعيد نتيجة في الإجابة البنية resp. إن حقل Body في resp يحتوي على استجابة الخادم كتدفق قابل للقراءة. بعد ذلك، تقرأ ioutil.ReadAll الإجابة بأكملها، وتُخزن النتيجة في b. إن تدفق Body يُغلق لتجنب أي تسرب للموارد، ويكتب Printf الإجابة على المخرج القياسي.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://gopl.io
<html>
<head>
<title>The Go Programming Language</title>
...
```

لو فشل طلب HTTP، فإنه يخرج تقرير الفشل بدلًا من ذلك:

```
$ ./fetch http://bad.gopl.io
fetch: Get http://bad.gopl.io: dial tcp: lookup bad.gopl.io: no such host
```

في كلتا حالتيّ الخطأ، يجعل os.Exit(1) العملية تخرج برمز الحالة 1.

تمرين 1.7: إن الوظيفة المسماة io.Copy(dst, src) تقرأ من src، وتكتب على dst. استخدمها بدلًا من ioutil.ReadAll لنسخ جسم الإجابة إلى os.Stdout دون طلب تخزين مؤقت كبير بما يكفي للإبقاء على التدفق الكامل. تأكد من فحص نتيجة خطأ io.Copy.

تمرين 1.8: عدّل برنامج fetch لإضافة اللاحقة http:// لكل عنوان معطى حتى لو كانت مفقودة. قد ترغب في استخدام strings.HasPrefix.

تمرين 1.9: عدّل فيتش لطباعة رمز HTTP أيضًا، والموجود في res.Status.

1.6 جلب العناوين بشكل متزامن

إن واحد من أحدث الجوانب وأكثر إثارة للاهتمام في Go هو دعمها للبرمجة المتزامنة. هذا موضوع كبير سنخصص له الفصلين 8 و 9، لذا سنقدم لك الآن مذاقًا من آليات تزامن Go الأساسية وروتينات Go وقنواتها.

إن البرنامج التالية، هو برنامج إحضار الكل أو fetchall، يقوم أيضًا بإحضار محتويات URL مثل المثال السابق، ولكنه يحضر العديد من العناوين ولكن بشكل متزامن، لذا لن تستغرق العملية وقت أطول من أطول عملية إحضار وليس مجموع عمليات الإحضار كلها. إن هذه النسخة من fetchall تتخلص من الاستجابات، ولكنها تقدم تقرير بحجم كل واحدة منها والوقت المستغرق فيها:

```
gopl.io/chl/fetchall
// Fetchall fetches URLs in parallel and reports their times and sizes.
package main
import (
	"fmt"
	"io"
	"io/ioutil"
	"net/http"
	"os"
	"time"
)
func main() {
	start := time.Now()
	ch := make(chan string)
	for _, url := range os.Args[1:] {
		go fetch(url, ch) // start a goroutine
	}
	for range os.Args[1:] {
		fmt.Println(<-ch) // receive from channel ch
	}
	fmt.Printf("%.2fs elapsed\n", time.Since(start).Seconds())
}
func fetch(url string, ch chan<- string) {
	start := time.Now()
```

```
    resp, err := http.Get(url)
    if err != nil {
        ch <- fmt.Sprint(err) // send to channel ch
        return
    }
    nbytes, err := io.Copy(ioutil.Discard, resp.Body)
    resp.Body.Close() // don't leak resources
    if err != nil {
        ch <- fmt.Sprintf("while reading %s: %v", url, err)
        return
    }
    secs := time.Since(start).Seconds()
    ch <- fmt.Sprintf("%.2fs  %7d  %s", secs, nbytes, url)
}
```

إليك مثال على هذا:

```
$ go build gopl.io/chl/fetchall
$ ./fetchall https://golang.org http://gopl.io https://godoc.org
0.14s    6852  https://godoc.org
0.16s    7261  https://golang.org
0.48s    2475  http://gopl.io
0.48s elapsed
```

إن روتين-جو (goroutine) هو تنفيذ وظيفة التزامن، والقناة (Channel) هي آلية تواصل تسمح بتمرير روتين-جو واحد لقيمة نوع محدد إلى روتين-جو آخر. إن الوظيفة main تعمل في روتين-جو ، وتخلق عبارة go المزيد من جورتين.

تصنع الوظيفة main قناة من السلاسل النصية باستخدام make. وبالنسبة لكل معطى في سطر الأوامر، تطلق عبارة go في حلقة التكرار الأولى روتين-جو جديدا يقوم باستدعاء وظيفة fetch بشكل غير متزامن لإحضار URL باستخدام http.Get. تقرأ وظيفة io.Copy جسم الاستجابة وتتخلص منه من خلال الكتابة على تدفق مُخرج ioutil.Discard. يعيد Copy حساب البِت، بجانب أي خطأ حدث. ومع وصول كل نتيجة، سترسل fetch ملخصا في القناة ch. إن حلقة التكرار الثانية في الوظيفة main تستقبل وتطبع هذه السطور.

عندما يحاول أحد روتين-جو إرسال أو استقبال في قناة ما، فإنه يقوم بالحجب حتى يحاول روتين-جو آخر بعملية الإرسال والاستقبال المناظرة، وعند هذه النقطة، تنقل القيمة، ويكمل كلا روتين-جو عملهما. وفي هذا المثال، يرسل كل fetch قيمة (ch <- expression) في القناة ch، وتستقبلهم الوظيفة الرئيسية كلهم (<-ch). إن جعل الوظيفة الرئيسية تقوم بكل الطباعة يضمن أن المُخرج من كل روتين-جو يعالج كوحدة واحدة، من دون خطر التعارض ما لو انتهى كلا روتينين جو في وقت واحد.

تمرين 1.10: جِد موقع ويب ينتج كمية كبيرة من البيانات، وافحص عملية الإحضار من خلال تشغيل fetchall مرتين متتابعين لترى هل الأوقات المذكورة في التقرير ستتغير بشكل كبير أم لا. هل حصلت على نفس المحتوى في كل مرة؟ عدّل fetchall لطباعة مُخرجه على ملف بحيث يمكن فحصه.

تمرين 1.11: جرب fetchall مع قوائم معطيات أطول، مثل عينات من أعلى مليون موقع متاحة على موقع alexa.com. كيف يتصرف البرنامج لو لم يستجيب الموقع وحسب؟ (يصف القسم 8.9 آليات التعامل في مثل هذه الحالات).

1.7 خادم الويب (Web Server)

تُسهّل مكتبات Go كتابة خادم ويب يستجيب لطلبات العميل الشبيهة بالطلبات التي يقدمها fetch. وسنقدم لك في هذا القسم خادم بمواصفات دنيا يعيد مكونات مسار ال URL المستخدم للوصول إلى الخادم. بمعنى، أنه لو كان الطلب على http://local-host:8000/hello فإن الاستجابة ستكون "URL.Path = "/hello.

```
gopl.io/ch1/server1
// Server1 is a minimal "echo" server.
package main
import (
    "fmt"
    "log"
    "net/http"
)
func main() {
    http.HandleFunc("/", handler) // each request calls handler
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
// handler echoes the Path component of the requested URL.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
```

إن البرنامج يتكون من حفنة من السطور فقط لأن وظائف المكتبة تقوم بمعظم العمل. وتربط الوظيفة الأساسية main وظيفة المداوِل (handler) مع الـ URLs القادمة التي تبدأ بـ /، أي كل الـ URLs الموجودة، وتبدأ تشغيل خادم يستمع للطلبات القادمة من المنفذ 8000. يُمثَّل الطلب كـ struct من النوع http.Request، والذي يحتوي على عدد من الحقول المرتبطة، أحدهما هو URL الطلب القادم. عند وصول طلب، يُعطى لوظيفة المداول، والتي تسحب عنصر المسار (/hello) من URL الطلب، وتعيدها كاستجابة، باستخدام fmt.Fprintf. سنشرح خوادم الويب بالتفصيل في القسم 7.7.

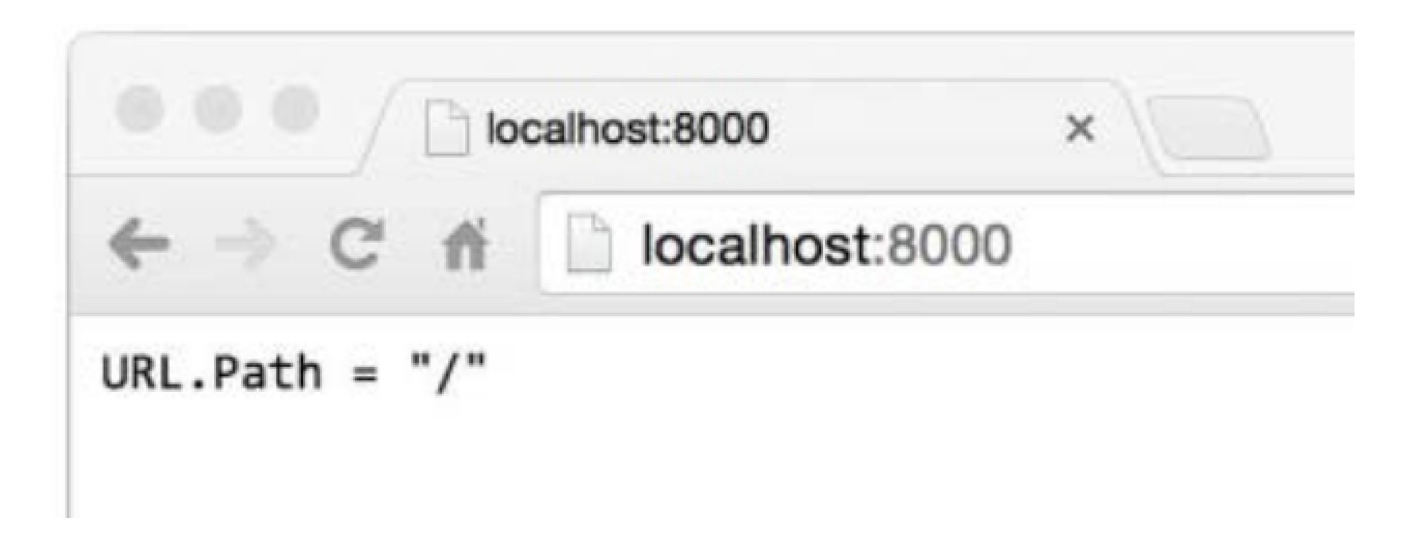
لنبدأ تشغيل الخادم في الخلفية. عند استخدام Mac OS أو لينوكس، قم بإضافة حرف العطف (&) للأمر، أما في ميكروسوفت ويندوز، ستحتاج لتشغيل الأمر بدون حرف عطف في نافذة أمر منفصلة.

```
$ go run src/gopl.io/ch1/server1/main.go &
```

ومن ثم يمكننا صنع طلبات العملاء من سطر الأوامر بالشكل التالي:

```
$ go build gopl.io/chl/fetch
$ ./fetch http://localhost:8000
URL.Path = "/"
$ ./fetch http://localhost:8000/help
URL.Path = "/help"
```

بدلًا من ذلك، يمكننا الدخول للخادم من متصفح الويب كما هو موضح في الشكل 1.2:



الشكل 1.2: استجابة من خادم الصدى.

من السهل إضافة خصائص إلى الخادم. وأحد الإضافات المفيدة هي URL محدد يعيد حالة من نوع ما. كمثال، تقوم هذه النسخة بنفس الصدى ولكنها تحسب عدد الطلبات أيضًا، وطلب URL/count يقدم الحساب حتى الآن ويستثني طلبات الحساب نفسها:

```
gopl.io/ch1/server2
// Server2 is a minimal "echo" and counter server.
package main
import (
    "fmt"
    "log"
    "net/http"
    "sync"
)
var mu sync.Mutex
var count int
func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/count", counter)
```

```
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
// handler echoes the Path component of the requested URL.
func handler(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    count++
    mu.Unlock()
    fmt.Fprintf(w, "URL.Path = %q\n", r.URL.Path)
}
// counter echoes the number of calls so far.
func counter(w http.ResponseWriter, r *http.Request) {
    mu.Lock()
    fmt.Fprintf(w, "Count %d\n", count)
    mu.Unlock()
}
```

يحتوي الخادم على اثنين من المداولين، ويحدد URL الطلب أيها يتم طلبه: إن طلب /count يستدعي العداد counter، بينما كل الطلبات الأخرى تستدعي المداوِل handler. إن نمط المداول الذي ينتهي بـ / يطابق أي URL يحتوي على النمط كسابقة له. يقوم الخادم وراء الكواليس بتشغيل المداول لكل طلب قادم في روتين-جو منفصل بحيث يمكنه خدمة طلبات متعددة بالتزامن. ولكن لو كان هناك طلبين متزامنين يحاولان تحديث الحساب في نفس الوقت، فقد لا يزيد بشكل متسق، وسيعاني البرنامج من عيب خطير اسمه race condition (انظر 9.1)، ولتجنب هذه المشكلة، يجب أن نضمن ألا يدخل أكثر من روتين-جو واحد فقط للمتغير في المرة الواحد، وهذا هو الهدف من المطالبة بـ mu.Lock() و mu.Unlock() الذين يضعا أقواس حول كل وصول لـ count. سنبحث التزامن بدقة أكبر عند الحديث عن المتغيرات المشتركة في الفصل التاسع.

المثال الأكثر ثراءً على هذا، هو أن وظيفة المداول يمكن أن تقدم تقرير حول الترويسات (headers) وبيانات الشكل التي تتلقاها، مما يجعل الخادم مفيد في طلبات الفحص وعلاج العيوب.

```
gopl.io/ch1/server3
// Server3 is an "echo" server that displays request parameters.
package main
import (
    "fmt"
    "log"
    "net/http"
)
func main() {
    http.HandleFunc("/", handler)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
//!+handler
// handler echoes the HTTP request.
func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s %s %s\n", r.Method, r.URL, r.Proto)
    for k, v := range r.Header {
```

```
        fmt.Fprintf(w, "Header[%q] = %q\n", k, v)
    }
    fmt.Fprintf(w, "Host = %q\n", r.Host)
    fmt.Fprintf(w, "RemoteAddr = %q\n", r.RemoteAddr)
    if err := r.ParseForm(); err != nil {
        log.Print(err)
    }
    for k, v := range r.Form {
        fmt.Fprintf(w, "Form[%q] = %q\n", k, v)
    }
}
```

يستخدم هذا حقول بنية http.Request لإنتاج مخرج كهذا:

```
GET /?q=query HTTP/1.1
Header["Accept-Encoding"] = ["gzip, deflate, sdch"]
Header["Accept-Language"] = ["en-US,en;q=0.8"]
Header["Connection"] = ["keep-alive"]
Header["Accept"] = ["text/html,application/xhtml+xml,application/xml;..."]
Header["User-Agent"] = ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_5)..."]
Host = "localhost:8000"
RemoteAddr = "127.0.0.1:59911"
Form["q"] = ["query"]
```

لاحظ كيف أن استدعاء ParseForm متداخل في عبارة if. تسمح لغة Go بعبارة بسيطة مثل إعلان متغير محلي بأن تسبق شرط if، وهو أمر مفيد بشكل خاص في التعامل مع الخطأ في هذا المثال. يمكننا كتابته كالتالي:

```
err := r.ParseForm()
if err != nil {
     log.Print(err)
}
```

ولكن دمج العبارات أقصر ويقلل نطاق المتغير err، وهو من الممارسات الجيدة. سنعرف النطاق في القسم 2.7.

لقد رأينا في هذه البرامج ثلاثة أنواع مختلفة جدًا استُخدمت كتدفقات الخرج. وقد نسخ برنامج fetch بيانات استجابة HTTP لـ os.Stdout، وهو ملف، وكذلك فعل برنامج lissajous. إن برنامج fetchall يلقي بالاستجابة بعيدًا (ولكنه يحسب طولها) من خلال نسخها إلى ioutil. Discard. واستخدم خادم الويب المذكور أعلاه fmt.Fprintf للكتابة إلى http.ResponseWriter ممثلًا متصفح الويب.

وبالرغم من اختلاف الأنواع الثلاثة في تفاصيل ما تقوم به، إلا أنها كلهم يستجيبون لواجهة (interface) مشتركة، تسمح باستخدام أي منهم كلما كان هناك حاجة إلى تدفق مُخرج. سنناقش هذه الواجهة، التي تُسمى io.Writer، في القسم 7.1.

إن آلية الواجهات في جو هي موضوع الفصل السابع، ولكننا سنعطيك فكرة عما يمكنها فعله، لنرى مدى سهولة دمج خادم الويب مع وظيفة lissajous، بحيث يمكن كتابة GIFs المتحركة على المخرج القياسي وليس على وظيفة lissajous. قم بإضافة هذه السطور إلى خادم الويب وحسب:

```
handler := func(w http.ResponseWriter, r *http.Request) {
    lissajous(w)
}
http.HandleFunc("/", handler)
```

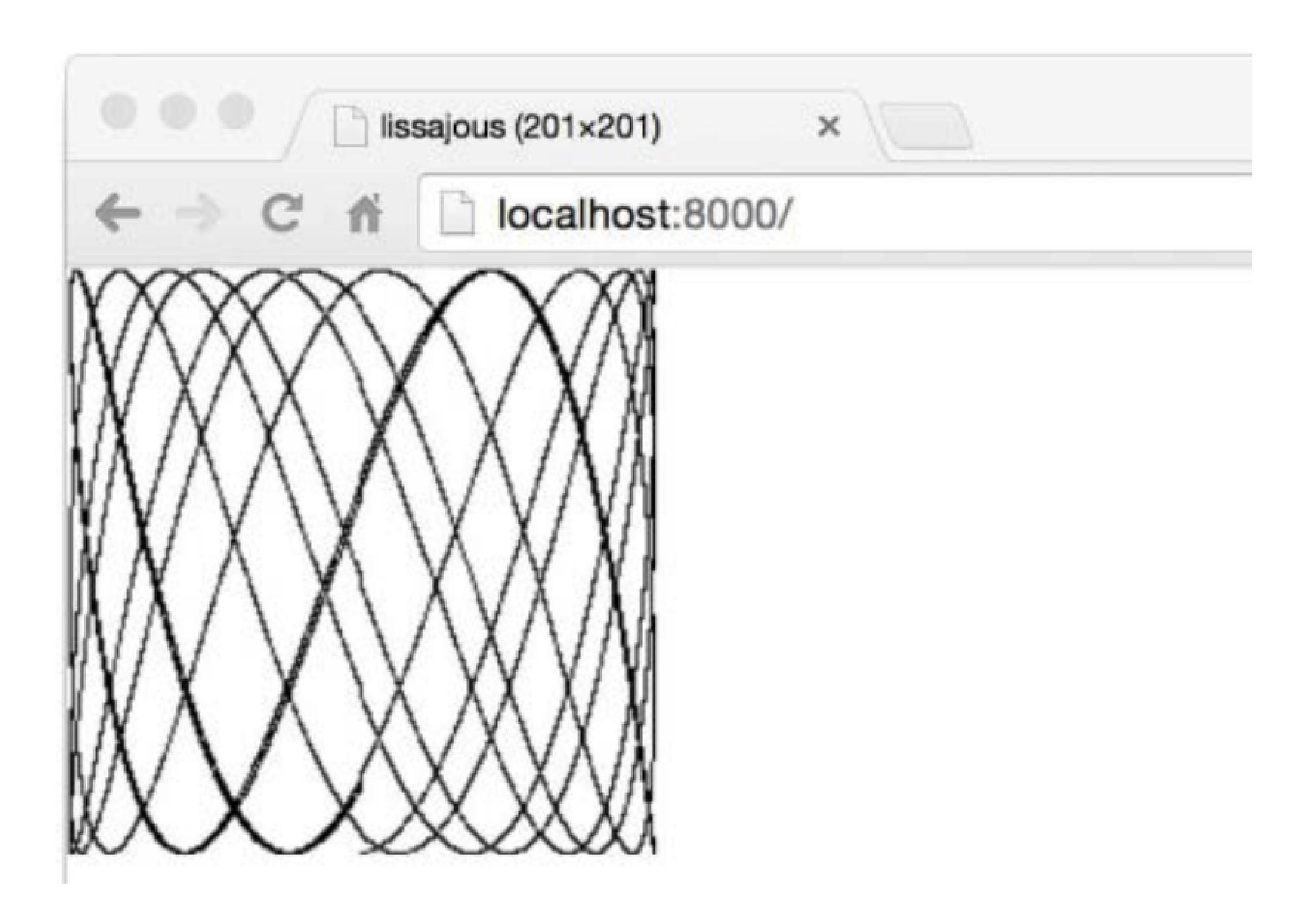
أو بشكل متساو:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request){
    lissajous(w)
})
```

إن المعطى الثاني في استدعاء وظيفة HandleFunc المذكورة أعلاه هو "القيمة الحرفية للوظيفة" (function literal)، وهي الوظيفة المجهولة التي تُحدد عند استخدامها فقط. سنشرح هذا بتفصيل أكثر في القسم 5.6.

بمجرد أن تقوم بهذا التغيير، قم بزيارة http://localhost:8000 في متصفحك. وفي كل مرة تحمل فيها الصفحة، سترى صورة متحركة كتلك الموجودة في الشكل 1.3.

تمرين 1.12: عدّل خادم Lissajous لقراءة قيم المعلمات من URL. كمثال، يمكنك ترتيب الأمر بحيث يمكن لـ URL مثل http://localhost:8000/?cycles=20 أن يحدد عدد الدورات ب 20 دورة بدلًا من العدد المعتاد وهو 5 دورات. استخدم وظيفة strconv.Atoi لتحويل معلمة السلسلة النصية إلى رقم صحيح. يمكنك رؤية توثيقها في مستند go: strconv.Atoi.



الشكل 1.3: صور Lissajous المتحركة في المتصفح.

1.8 النهايات المفككة

تحتوي Go على أشياء أكثر مما غطيناها في هذه المقدمة السريعة بكثير، وإليك بعض المواضيع التي لم نتطرق لها إلا بشكل عابر أو لم نذكرها على الإطلاق، ولكننا سنقدم نقاش موجز عنها لجعلها مألوفة عندما تظهر بشكل مختصر قبل المعالجة الكاملة لها.

تدفق التحكم (Control Flow): لقد تحدثنا عن عبارتيّ تدفق تحكم أساسيتين، وهما if و for، ولكننا لم نتحدث عن عبارة switch، وهي فرع متعدد المسارات. وإليك مثال صغير عليها:

```
switch coinflip() {
case "heads":
    heads++
case "tails":
    tails++
```

```
default:
    fmt.Println("landed on edge!")
}
```

إن نتيجة استدعاء coinflip تُقارن مع نتيجة كل حالة. وتقيم الحالات من القمة إلى القاع، بحيث تنفذ أول حالة بها تطابق. إن الحالة الاعتيادية الاختيارية ستطبق ما لم تتطابق أي من الحالات الأخرى، ويمكن وضعها في أي مكان. لا تتابع الحالات من واحدة إلى الأخرى كما هو الحال في لغات C واللغات الشبيهة بها (بالرغم من وجود عبارة fallthrough تتجاوز هذا السلوك ولكنها نادرًا ما تُستخدم).

لا يحتاج switch إلى معامل، بل يمكنه وضع قائمة بالحالات وحسب، حيث يعتبر كل منها تعبير منطقي (Boolean):

```
func Signum(x int) int {
    switch {
    case x > 0:
        return +1
    default:
        return 0
    case x < 0:
        return -1
    }
}
```

يطلق على هذا الشكل (tagless switch)، وهو يكافئ switch true.

كما هو الحال في عبارات for و if، يمكن أن يتضمن switch عبارة اختيارية بسيطة – إعلان قصير عن متغير، أو عبارة زيادة أو تخصيص، أو نداء وظيفة – يمكن استخدامها لتحديد قيمة قبل اختبارها.

إن عبارات break و continue تعدل تدفق التحكم. وتجعل عبارة break التحكم يستمر في العبارة التالية بعد عبارة for أو switch أو select الداخلية (وهو ما سنراه لاحقًا)، وكما رأينا في القسم 1.3، أما continue تجعل حلقة for الداخلية تبدأ تكرارها التالي. يمكن تسمية العبارات بحيث يمكن لـ break و continue أن يشيروا لهم، كمثال، الخروج من الحلقات المتداخلة فورًا، أو بدء التكرار التالي للحلقة الخارجية. حتى أنه توجد عبارة goto ، بالرغم من أنها خاصة بالشفرة التي تنتجها الآلة، وليس الاستخدام المنتظم من قبل المبرمجين.

الأنواع المُسمّاة Named typed: إن الإعلان عن النوع يجعل من الممكن منح اسم لنوع موجود بالفعل، وحيث أن أنواع البنيات struct عادة ما تكون طويلة، فإنها تُسمى دائمًا تقريبًا. إن المثال المألوف على هذا هو تعريف نوع النقطة Point الخاصة بنظام الرسم ثنائي الأبعاد:

```
type Point struct {
    X, Y int
}
```

```
var p Point
```

سنتحدث في الفصل الثاني عن الإعلانات عن النوع والأنواع المسماة.

المؤشرات (Pointers): تقدم Go مؤشرات، وهي القيم التي تحتوي على عنوان المتغير. وفي بعض اللغات، وخاصة لغة C، تكون القيود قليلة نسبيًا على المؤشرات. بينما في بعض اللغات الأخرى، تتخفى المؤشرات كـ "مراجع"، ولا يوجد الكثير مما يمكن عمله فيهم إلا تمريرهم وحسب. تقع Go في منطقة ما في الوسط بين هذين النوعين من اللغات، فالمؤشرات فيها ظاهرة بوضوح، ومعامل & ينتج عنه عنوان المتغير، ومعامل * يستعيد المتغير الذي يشير له المؤشر، ولكن لا يوجد عملية حسابية للمؤشر. سنشرح المؤشرات بالتفصيل في القسم 2.3.2.

الطرق والواجهات Methods & interfaces: إن الطريقة هي وظيفة ترتبط مع نوع مُسمى، ولكن Go غير معتادة في كون طرقها قد تكون متصلة بأي نوع مسمى تقريبًا. سنتحدث عن الطرق في الفصل السادس. أما الواجهات فهي أنواع مجردة تجعلنا نعامل الأنواع الملموسة المختلفة بنفس الطريقة بناء على الطرق التي تحتويها، وليس بالكيفية التي يتم تمثيلها أو تطبيقها بها. إن الواجهات هي موضوع الفصل السابع.

الحِزم Packages: تأتي لغة Go مع مكتبة قياسية شاملة من الحزم المفيدة، وقد صنع مجتمع Go وشارَك الكثير غيرها. إن البرمجة عادة ما تدور حول استخدام الحزم الموجودة أكثر من حول كتابة شفرة أصلية خاصة بالفرد. وسنوضح في هذا الكتاب دستتين من أهم الحزم القياسية، ولكن هناك حزم أكثر بكثير غيرها ليس لدينا مساحة كافية للحديث عنها، ولا يمكننا تقديم مرجع كامل لأي حزمة.

قبل أن نباشر العمل على أي برنامج جديد، سيكون من الجيد أن نرى هل توجد حزم موجودة بالفعل يمكنها مساعدتنا على إنهاء عملنا بسهولة أكبر أم لا. يمكنك إيجاد فهرس لحزم المكتبة القياسية على https://golang.org/pkg، والحزم التي ساهم بها المجتمع على https://godoc.org. إن أداة go doc تجعل الوصول لهذه المعلومات سهلًا من خلال سطر الأوامر:

```
$ go doc http.ListenAndServe
package http // import "net/http"
func ListenAndServe(addr string, handler Handler) error
    ListenAndServe listens on the TCP network address addr and then
    calls Serve with handler to handle requests on incoming connections.
...
```

التعليقات: لقد ذكرنا بالفعل التعليقات التوثيقية في بداية البرنامج أو الحزمة. من الجيد أيضًا كتابة تعليق قبل الإعلان عن كل وظيفة من أجل تحديد سلوكها. إن هذه التقاليد مهمة لأنها تُستخدم عن طريق أدوات مثل go doc و godoc لتحديد موقع المستندات وعرضها (انظر 10.7.4).

بالنسبة للتعليقات التي تضم سطور متعددة أو تظهر داخل تعبير أو عبارة، يوجد أيضًا تدوين /*...*/ المألوف من اللغات الأخرى. وتُستخدم هذه التعليقات أحيانًا في بداية الملف في فقرة النص التفسيري الطويلة لتجنب // في كل سطر.

إن // و /* داخل التعليق ليس لهم معنى خاص، وبالتالي لا تتداخل التعليقات بسببهم.

2- بنية البرنامج

تستطيع من خلال لغة البرمجة جو (Go) كمثيلاتها من لغات البرمجة بناء برنامج ضخم بواسطة مجموعة صغيرة من التعليمات البرمجية الأساسية، ويتم ذلك من خلال حفظ قيم بداخل المتغيرات (Variables) وكتابة مجموعة من التعابير البرمجية (expressions) وربطها مع بعضها البعض من خلال عمليات متنوعة مثل الجمع والطرح، و تجميع الأنماط الأساسية من خلال وضعها في مصفوفة (array) أو بنية واحدة. تستخدم التعابير البرمجية (Expressions) في الجمل البرمجية التي تنفذ حسب ترتيب معين يتحكم به عن طريق تعابير تدفق التحكم من مثل مثل جملة If أو جملة for، وتجمع الجمل البرمجية ضمن دوال (Functions) لفصل الجمل عن بعضها البعض ومن أجل إعادة استخدامها لاحقًا، وتجمع تلك الدوال ضمن ملفات المصدر (source files) أو الحُزَم.

لقد عرضنا العديد من الأمثلة على الدوال والجمل البرمجية والتعبيرية خلال الفصل السابق، في هذا الفصل سنتعلم المزيد عن العناصر البنائية الأساسية للغة البرمجة جو، وسنشرح أمثلة توضيحية لبرامج بسيطة غير معقدة وذلك للتركيز على أساسيات اللغة دون الخوض في تعقيدات الخوارزميات وهياكل البيانات.

2.1 الأسماء Names

في لغة البرمجة جو تتبع الأسماء التي تُطلق على الدوال والمتغيرات والثوابت والأنماط ومسميات الجمل والحُزم قاعدة بسيطة وهي: يجب أن يبدأ الاسم بحرف (أي حرف طبقًا لأنظمة الترميز العالمية Unicode) أو تسطير سفلي (_) (Underscore)، ويمكن وضع أرقام والتسطير السفلي في منتصف الاسم أو آخره . ويجب عليك معرفة أن الأحرف الكبيرة تختلف عن الأحرف الصغيرة، على سبيل المثال heapsort و Heapsort هما اسمان مختلفان.

يوجد في لغة جو 25 كلمة مفتاحية لا يمكن استخدامها كأسماء مثل if و switch لأن هذه الأسماء لها وظائف مخصصة، وهذه الكلمات هي:

```
break       default      func     interface  select
case        defer        go       map        struct
chan        else         goto     package    switch
const       fallthrough  if       range      type
continue    for          import   return     var
```

بالإضافة للأسماء السابقة، توجد ثلاثة مجموعات من الأسماء المعلنة سابقًا (predeclared names) مثل int و true، فهذه الأسماء محجوزة للثوابت والأنماط والدوال الخاصة بالبرنامج، وهي كالآتي:

الثوابت: true false iota nil

الأنماط: int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

float32 float64 complex128 complex64

bool byte rune string error

الدوال: make len cap new append copy close delete

complex real imag

panic recover

هذه الأسماء ليست محجوزة للبرنامج، وبالتالي بإمكانك استخدامها في إعلان الجمل، سنرى العديد من الأمثلة التي توضح استخدام الأسماء المعلنة ولكن كن حذرًا كونها محيرة نوعًا ما.

في حال إعلان الكيان (entity) ضمن الدالة فإن ذلك الكيان يصبح محلي (local) أي ينتمي لتلك الدالة، وفي حال إعلان الكيان خارج الدالة سيصبح الكيان مرئي في جميع ملفات الحزمة التي ينتمي إليها الكيان.إن حالة الحرف الأول من الأسم تحدد مجال رؤيته في حدود الحزم، فإذا كان الحرف الأول للاسم هو حرف كبير فإن الاسم مُصدّر (exported)، والمقصود بمُصدّر هنا أن الاسم مرئي ويمكن الوصول إليه خارج حدود حزمته ويمكن الإشارة إليه من خلال أجزاء البرنامج الأخرى مثل Printf المستخدمة في حزمة fmt، لاحظ أن أسماء الحزم دائمًا تبدأ بحرف صغير.

لا يوجد حدود لطول الاسم، ولكن يفضل أن تبقى الأسماء في لغة جو قصيرة كونها أكثر تناسقًا وأناقة، وخصوصًا للمتغيرات المحلية ذات النطاقات الصغيرة، وربما سترى كثيرا متغيرات سميت بـ I بدلا عن theLoopIndex. وبشكل عام، ي كلما كبر نطاق الاسم زاد طوله ودل على معناه.

من الناحية الجمالية، يستخدم مبرمجي جو أسلوب الدمج النصفي (camel case) عندما يخترعون الأسماء، وهي تفضيل أن يكون بداية الكلمة الثانية حرف كبير بدلا عن التسطير السفلي. بالتالي ستجد في المكتبات القياسية دوال بأسماء مثل QuoteRuneToASCII و parseRequestLin ولكن لن تجد أسماء مفصولة بتسطير سفلي مثل quote_rune_to_ASCII أو parse_request_line، أما حروف الاختصارات و الحروف الأولى من كل كلمة مثل ASCII و

HTML فيجب أن تكون كلها بنفس الهيئة، وبالتالي من الممكن تسمية الدالة htmlEscape أو HTMLEscape أو escapeHTML ولكن لا يمكن تسميتها escapeHtml.

2.2 الإعلانات Declarations

الإعلان يستخدم لتحديد كيان البرنامج وتحديد بعض أو جميع خصائصه، يوجد أربعة أنواع أساسية للإعلانة وهي: المتغيرات var والثوابت const والأنماط type والدوال func، سنتحدث عن المتغيرات والأنماط ضمن هذا الفصل، وسنتحدث عن الثوابت في الفصل الثالث والدوال في الفصل الخامس.

تُحفظ برامج جو في ملفات ينتهي اسمها بـ (go.)، وكل ملف يبدأ بإعلان الحزمة package للإشارة عن ماهية الحزمة، ويتبع إعلانة الحزمة package أي إعلانات استيراد import ومن ثم سلسلة من إعلانات على مستوى الحزمة للأنماط والمتغيرات والثوابت والدوال، علمًا أنه لا يشترط الترتيب. على سبيل المثال يشير هذا البرنامج إلى ثابت ودالة ومتغيرين:

```
gopl.io/ch2/boiling
// Boiling prints the boiling point of water.
package main
import "fmt"
const boilingF = 212.0
func main() {
    var f = boilingF
    var c = (f - 32) * 5 / 9
    fmt.Printf("boiling point = %g°F or %g°C\n", f, c)
    // Output:
    // boiling point = 212°F or 100°C
}
```

الثابت boilingF يمثل إعلان على مستوى الحزمة (وكذلك main)، بينما المتغير f و c هما محليان بالنسبة للدالة main، إن اسم كل كيان على مستوى الحزمة مرئي ليس فقط من خلال ملف المصدر الذي يحتوي على إعلان الكيان، بل مرئي من خلال جميع الملفات في الحزمة. على النقيض من ذلك، الإعلان المحلي مرئي فقط ضمن الدالة التي تُعلن فيها وربما فقط ضمن جزء صغير منها.

فيما يتعلق بإعلانة الدالة فإنها تحتوي على اسم وقائمة من المعاملات (المتغيرات التي تزودها بالقيم من خلال مستدعيي الدالة callers وقائمة اختيارية من النتائج ونص الدالة، الذي بدوره يحتوي على الجمل البرمجية التي تحدد وظيفة الدالة. وتستبعد قائمة النتائج إذا كانت الدالة لا ترجع شيئا. يبدأ تنفيذ الدالة من الجملة الأولى ويستمر حتى

يصل إلى جملة عودة return أو عند وصوله إلى نهاية الدالة التي لا تحتوي على نتائج، وعند الحصول على النتائج يتم إرجاعها إلى المستدعي caller.

لقد شاهدنا العديد من الوظائف وهناك المزيد من الوظائف التي سنراها لاحقًا وسنشرح الوظائف بشكل مفصل في الفصل الخامس، ولكن سنشرح الآن عن الوظائف بشكل عام لتفهمها بشكل مبسط، الدالة fToC الموضحة في البرنامج أدناه تحتوي على معادلة تحويل الحرارة من فهرنهايت إلى مئوية والعكس، وبالتالي فقد عرفت لمرة واحدة ولكن يمكن استخدامها في أكثر من موضع، ونلاحظ أن main قد استدعتها مرتين باستخدام قيمة ثابتين محليين مختلفين:

```
gopl.io/ch2/ftoc
// Ftoc prints two Fahrenheit-to-Celsius conversions.
package main
import "fmt"
func main() {
    const freezingF, boilingF = 32.0, 212.0
    fmt.Printf("%g°F = %g°C\n", freezingF, fToC(freezingF)) // "32°F = 0°C"
    fmt.Printf("%g°F = %g°C\n", boilingF, fToC(boilingF))   // "212°F = 100°C"
}
func fToC(f float64) float64 {
    return (f - 32) * 5 / 9
}
```

2.3 المتغيرات Variables

يستخدم إعلان المتغير var لإنشاء متغيرات لنمط معين وتحدد اسمًا له وتحدد قيمته الأولية، كل إعلان له الصيغة العامة التالية:

```
var name type = expression
```

يمكن الاستغناء عن أو حذف النوع (type) أو التعبير (=expression) ولكن لا يمكن حذفهما كلاهما معًا، في حال حذف النمط فإنه سيحدد من خلال التعبير المبدئي، وفي حال حذف التعبير فإن قيمة النمط المبدئية ستكون صفر (zero value والتي يرمز لها بـ 0 في حال الأرقام وبخطأ (false) في الجمل المنطقية (booleans) وبإشارة اقتباس ("") في السلاسل وبـ nil في الواجهات والأنواع المرجعية (شريحة، مؤشر، خريطة، قناة، دالة). والصفر في حالة الأنماط المركبة مثل المصفوفات أو البنية يكون من خلال وضع قيمة صفر في جميع عناصر المصفوفة أو الحقول.

الغاية من استخدام آلية الصفر مع المتغيرات هي لضمان تخزين قيمة معرفة في المتغير حسب نمطه، ففي لغة البرمجة جو لا يوجد شيء اسمه متغير غير معد مبدئيا، وذلك يبسط الكود ويضمن عادة سلوكا معقولا للشروط الحدية من دون الحاجة للمزيد من العمل، على سبيل المثال:

```
var s string
fmt.Println(s) // ""
```

يطبع سلسلة فارغة (سطر فارغ)، بدلا من التسبب بخطأ ما أو بسلوك غير متوقع، غالبًا ما يقوم مبرمجو جو ببذل جهدا لجعل القيمة الصفرية ذات معنى في الأنواع المعقدة، وبالتالي تبدأ المتغيرات حياتها بحالة مفيدة.

من الممكن إعلان أو وضع قيمة مبدئية لمجموعة من المتغيرات ضمن جملة إعلان واحد بواسطة استخدام مجموعة من التعابير الملائمة، كما نلاحظ أن حذف النمط يتيح إمكانية إعلان عن مجموعة من المتغيرات ذات الأنماط المختلفة:

```
var i, j, k int                 // int, int, int
var b, f, s = true, 2.3, "four" // bool, float64, string
```

من الممكن أن تكون القيم المبدئية عبارة عن قيم فعلية أو تعابير عشوائية، فالمتغيرات على مستوى الحزمة تهيء بالقيم المبدئية قبل بداية main (راجع القسم 2.6.2)، وتوضع قيم مبدئية في المتغيرات المحلية عند الوصول إلى إعلانها خلال تنفيذ الدالة.

ويمكن أيضاً تخزين قيم مبدئية في المتغيرات من خلال استدعاء دالة ترجع قيم متعددة:

```
var f, err = os.Open(name) // os.Open returns a file and an error
```

2.3.1 إعلانات المتغيرات القصيرة Short Variable Declarations

يمكن استخدام صيغة تسمى "إعلان المتغيرات القصيرة" ضمن الدالة لإعلان المتغيرات المحلية وتخزين قيم بداخلها، الصيغة العامة لإعلان المتغيرات القصيرة هي name := expression، ويتم تحديد نوع الاسم من خلال نمط التعبير، ها هنا ثلاثة إعلانيات لمتغيرات قصيرة متعددة في دالة lissajous (راجع القسم 1.4):

```
anim := gif.GIF{LoopCount: nframes}
freq := rand.Float64() * 3.0
t := 0.0
```

وكون إعلان المتغيرات القصيرة بسيطة وسهلة في التعامل فإنها تستخدم لإعلان وتحديد قيمة معظم المتغيرات المحلية. وعادة ما يستخدم إعلان var مع المتغيرات المحلية التي تحتاج إلى نمط محدد وواضح ليميزها عن التعابير المبدئية أو مع تلك المتغيرات التي لا تهم قيمتها الحالية وإنما ستستخدم لتخزين قيمة أخرى بها لاحقًا:

```
i := 100                  // an int
var boiling float64 = 100 // a float64
var names []string
var err error
var p Point
```

ومثل إعلان var، فإنه من الممكن إعلان عدة متغيرات وتخزين قيم بداخلهم من خلال جملة إعلانية واحدة للمتغيرات القصيرة:

```
i, j := 0, 1
```

ولكن عليك معرفة أن إعلان مجموعة من التعابير المبدئية معًا يستخدم فقط لتسهيل قراءة البرنامج، كما هو مع المجاميع الأساسية والقصيرة مثل وضع قيم مبدئية لجزء من حلقة for loop.

ويجب عليك ملاحظة أن الرمز := يستخدم للإعلان في حين الرمز = يستخدم لتعيين القيم. ويجب عليك التفريق بين إعلان مجموعة من المتغيرات وتعيين الصفوف (tuple assignment) ويسمى أيضًا بتعيين الحقول المترابطة (راجع القسم 2.4.1) حيث أنه تعين قيمة لكل متغير على الطرف الأيسر طبقًا للقيم الموجودة على الطرف الأيمن:

```
i, j = j, i // swap values of i and j
```

كما هو الحال في إعلان المتغيرات الاعتيادي var يمكن استخدام إعلان المتغيرات القصيرة لاستدعاء وظائف مثل os.Open حيث أن هذه الدالة تعيد قيمتين أو أكثر:

```
f, err := os.Open(name)
if err != nil {
    return err
}
// ...use f...
f.Close()
```

يجب عليك معرفة أنه ليس من الضرورة أن إعلان المتغيرات القصيرة يعلن جميع المتغيرات الموجودة على الطرف الأيسر، ففي حال كون إحدى المتغيرات قد أعلن عنه مسبقًا ضمن نفس المجموعة اللغوية (lexical block) (راجع القسم 2.7) فإن جملة إعلان المتغير ستعمل على أنها جملة تعيين لتلك المتغيرات.

انظر إلى الكود أدناه، ستلاحظ أن الجملة الأولى تعلن كلا من in و err في حين أن الجملة الثانية تعين قيمة في المتغير err كونه موجود سابقًا:

```
in, err := os.Open(infile)
// ...
out, err := os.Create(outfile)
```

يجب على إعلان المتغيرات القصيرة أن يعلن عن متغير واحد على الأقل وبالتالي فإن الكود التالي لن يترجم:

```
f, err := os.Open(infile)
// ...
f, err := os.Create(outfile) // compile error: no new variables
```

وللتخلص من هذه المشكلة يجب استخدام جملة التعيين التقليدية للجملة الثانية.

يحل إعلان المتغيرات القصيرة محل جملة التعيين فقط عندما تكون المتغيرات معلنه مسبقًا في نفس المجموعة اللغوية (lexical block)، وبالتالي فإن الإعلانات الموجودة خارج المجموعة ستتجاهل، سنرى أمثلة على ذلك في نهاية الفصل.

2.3.2 المؤشرات Pointers

المتغير هو عبارة عن مكان تخزين يحوي على قيمة ما، وتنشىء المتغيرات من خلال إعلان يحددها باسم محدد مثل x، ولكن هناك العديد من المتغيرات التي تحدد بتعبير مثل x[i] أو x.f، فجميع هذه التعابير تحدد قيمة المتغير ما لم تكن مكتوبة في الجانب الأيسر للجملة، وبالتالي في هذه الحالة ستعين قيمة جديدة للمتغير.

قيمة المؤشر pointer هي عنوان المتغير، وبالتالي فإن المؤشر عبارة عن موقع في الذاكرة، تخزن فيه القيمة، وليس كل قيمة لها عنوان وإنما كل عنوان له قيمة، الفائدة من المؤشر هي قراءة أو تحديث قيمة المتغير بشكل غير مباشر وذلك دون الحاجة حتى لمعرفة اسم المتغير، إذا كان في الحقيقة يملك اسما.

في حال إعلان عن المتغير على var x أنه من نوع int، فإن التعبير &x ("عنوان x") ينتج مؤشرا إلى متغير ذي عدد صحيح، وهذا العدد الصحيح هو قيمة النمط *int (وتقرأ مؤشر إلى int)، لو افترضنا أن القيمة تساوي p، فإننا نقرأها "p تؤشر إلى x" أو "p تحتوي على عنوان x". يكتب المتغير الذي تشير إليه p بالصورة التالية *p. وينتج التعبير *p قيمة ذلك المتغير int ولكن بما أن *p تشير إلى المتغير فقد تظهر أيضًا في الجهة اليسرى للجملة وبالتالي في هذه الحالة ستقوم الجملة بتحديث قيمة المتغير.

```
x := 1
p := &x          // p, of type *int, points to x
fmt.Println(*p) // "1"
*p = 2          // equivalent to x = 2
fmt.Println(x)  // "2"
```

كل عنصر من متغيرات النوع المجمع aggregate (أي عنصر من مصفوفة أو حقل من بنية) يعتبر متغيرًا أيضا وبالتالي له عنوان أيضًا.

قد توصف المتغيرات أيضًا على أنها قيم معنونة، والتعابير التي تشير المتغيرات هي التعابير الوحيدة التي ينطبق عليها عنوان المعامل &.

إن القيمة الصفرية في المؤشر هي nil، وتكون نتيجة الاختبار التالي p != nil صحيحة(true) في حال كانت p تؤشر إلى متغير، فالمؤشرات قابلة للمقارنة، فيتساوى مؤشران فقط عندما يؤشران إلى نفس المتغير أو إذا كان كلاهما صفرا nil.

```
var x, y int
fmt.Println(&x == &x, &x == &y, &x == nil) // "true false false"
```

إنه آمن بشكل كامل أن ترجع دالة ما عنوان متغير محلي، على سبيل المثال انظر للشفرة أدناه، ستجد أن المتغير v المحلي الذي أنشئ بواسطة استدعاء دالة f سيبقى موجودًا رغم انتهاء الاستدعاء، وسيبقى المؤشر p مؤشرًا إليه:

```
var p = f()
func f() *int {
     v := 1
     return &v
}
```

نلاحظ أن كل استدعاء لـ f يعود بقيمة مميزة:

```
fmt.Println(f() == f()) // "false"
```

ولأن المؤشر يحتوي على عنوان المتغير فإن المؤشر الذي يمرر كمعامل لدالة ما، فإنه يمكن أن يحدث المتغير الذي مرر بشكل غير مباشر. على سبيل المثال، الدالة التالية تزيد قيمة المتغير الذي يؤشر معاملها إليه ويرجع القيمة الجديدة للمتغير، لذا يمكن أن تستخدم في تعبير:

```
func incr(p *int) int {
     *p++ // increments what p points to; does not change p
     return *p
}
v := 1
incr(&v)              // side effect: v is now 2
fmt.Println(incr(&v)) // "3" (and v is 3)
```

في كل مرة نأخذ فيها عنوان المتغير أو ننسخ مؤشرا فإننا ننشئ مرادفات جديدة (aliases) أو طرق جديدة لتعريف نفس المتغير، على سبيل المثال p* هو عبارة عن مرادف لـ v. مرادفات المؤشرات مفيدة كونها تتيح لنا إمكانية الوصول للمتغير دون الحاجة إلى استخدام اسمه، ولكنها سيف ذو حدين لأن إيجاد جميع الجمل التي تصل لمتغير ما علينا أن نعرف جميع مرادفاته. ليس المؤشرات وحدها التي تنشئ المرادفات، إنما من الممكن أن تنشئ المرادف عند نسخ قيم أنماط مرجعية أخرى مثل الشرائح (slices) والخرائط (maps) والقنوات (channels) والبنيات (structs) والمصفوفات (arrays) والواجهات التي تحتوي تلك الأنماط.

المؤشرات هي المفتاح لحزمة flag التي تستخدم معاملات سطر الأوامر لتحديد قيم متغيرات معينة موزعة في البرنامج نفسه. لغاية التوضيح، التنوع الموجود في أمر الصدى echo السابقة تأخذ معاملين اختياريين: n- الذي يسبب في جعل أمر الصدى يحذف علامات السطر الجديد الزائدة التي كان من الطبيعي أن تطبع، و s sep- الذي يجعل أمر

الصدى يفصل المعاملات الخارجة طبقًا لمحتوى سلسلة الفصل sep بدلا من وضع مسافة واحدة افتراضية، وبما أن هذا هو الإصدار الرابع فإن الحزمة تسمى gopl.io/ch2/echo4.

```
gopl.io/ch2/echo4
// Echo4 prints its command-line arguments.
package main
import (
    "flag"
    "fmt"
    "strings"
)
var n = flag.Bool("n", false, "omit trailing newline")
var sep = flag.String("s", " ", "separator")
func main() {
    flag.Parse()
    fmt.Print(strings.Join(flag.Args(), *sep))
    if !*n {
        fmt.Println()
    }
}
```

تقوم الدالة flag.Bool بإنشاء متغير راية جديد من النمط bool وتأخذ ثلاثة معاملات: اسم الراية "n" وقيمة المتغير الافتراضية false والرسالة التي ستنطبع في حال قدّم المستخدم معامل خاطئ أو راية خاطئة أو h- أو help-. وبشكل مماثل يأخذ flag.String اسمًا وقيمة افتراضية ورسالة وينشئ متغير سلسلة نصية. المتغيران sep و n هما مؤشران إلى متغيرات الراية التي يجب الوصول إليها بطريقة غير مباشرة كما هو الحال مع sep* وn*.

عند تنفيذ البرامج سيستدعي flag.Parse قبل استخدام الرايات وذلك لتحديث متغيرات الرايات عن قيمهم الافتراضية، يمكن الحصول على معاملات بدون راية من خلال ()flag.Args على صورة شريحة من سلسلة نصية. إذا واجه flag.Parse خطأ ما فإنه يطبع رسالة سوء استخدام ويستدعي os.Exit(2) لينهي البرنامج.

دعنا نجري بعض التجارب على برنامج echo:

```
$ go build gopl.io/ch2/echo4
$ ./echo4 a bc def
a bc def
$ ./echo4 -s / a bc def
a/bc/def
$ ./echo4 -n a bc def
a bc def$
$ ./echo4 -help
Usage of ./echo4:
    -n    omit trailing newline
    -s    string separator (default " ")
```

2.3.3 دالة new

هناك طريقة أخرى لإنشاء متغير وهي استخدام الدالة المدمجة new، فإن التعبير new(T) ينشئ متغير غير مسمى نمطه T ويبدأه بقيمة صفر لـ T ويعيد عنوانه الذي هو قيمة النمط T*.

```
p := new(int)     // p, of type *int, points to an unnamed int variable
fmt.Println(*p)   // "0"
*p = 2            // sets the unnamed int to 2
fmt.Println(*p)   // "2"
```

لا يختلف المتغير الناشئ من دالة new عن أي متغير محلي عادي ذي عنوان محجوز عدا أنه لا يوجد هناك حاجة لاختراع (أو إعلان) عن اسم وهمي، ويمكننا استخدام new(T) في جملة تعبيرية، وبالتالي الدالة new جيدة فقط من الناحية النحوية وليس فكرتها الأساسية.

إن دالتي newInt أدناه لهما سلوكيات مماثلة:

```
func newInt() *int {
      var dummy int
      return &dummy
}
```

```
func newInt() *int {
      return new(int)
 }
```

كل استدعاء لدالة new ستسترجع متغيرات متميزة ذات عناوين فريدة:

```
p := new(int)
q := new(int)
fmt.Println(p == q) // "false"
```

هناك استثناء وحيد للقاعدة وهو أن أي متغيرين يكون نمطهما لا يحملان أية معلومة وبالتالي حجمهما صفر مثل struct{} أو int[0] فلهما نفس العنوان اعتمادا على كيفية برمجتهما.

نادرًا ما تستخدم دالة new كون معظم المتغيرات غير المسماه تكون من نوع البنية struct وبالتالي بناء الجملة الحرفي للبنية يكون أكثر مرونة (راجع القسم 4.4.1).

وكون الدالة new معلنة مسبقًا وليست كلمة مفتاحية فمن الممكن إعادة تعريف الاسم لشيء آخر ضمن الدالة، على سبيل المثال:

```
func delta(old, new int) int { return new - old }
```

وبالطبع الدالة new ضمن دالة delta ستكون غير متوفرة.

2.3.4 فترة بقاء المتغيرات Lifetime of Variables

المقصود بفترة بقاء المتغيرات (أو عمر المتغيرات) هو الفترة الزمنية التي يكون فيها المتغير موجودًا حين تنفيذ البرنامج، وعمر المتغيرات على مستوى الحزمة يكون طوال فترة تنفيذ البرنامج.

وعلى النقيض من ذلك تمتلك المتغيرات المحلية فترة بقاء آلية، أي تنشئ نسخة جديدة للمتغير في كل مرة تنفذ فيها جملة الإعلان ويستمر بقاء المتغير حتى يتعذر الوصول إليه، وفي هذه المرحلة يمكن إعادة استخدام حيز التخزين الخاص به. تعتبر معاملات الدوال والنتائج متغيرات محلية أيضًا حيث يعاد تشكيلهم في كل مرة يستدعى فيها دالتهم المضمنة.

على سبيل المثال أنظر أدناه إلى مقتطفات من برنامج ليساجوس (Lissajous) من القسم 1.4.

```
for t := 0.0; t < cycles*2*math.Pi; t += res {
   x := math.Sin(t)
   y := math.Sin(t*freq + phase)
   img.SetColorIndex(size+int(x*size+0.5), size+int(y*size+0.5),
      blackIndex)
}
```

ينشأ المتغير t في كل مرة تبدأ فيها حلقة for وتنشأ متغيرات جديدة x و y في كل تكرار للحلقة.

كيف يمكن لجامع النفايات أن يعرف أن حافظة المتغير يمكن استردادها؟ إن شرح القصة بالكامل هنا سيكون أكثر من اللازم هنا، ولكن الفكرة الأساسية هي أن كل متغير على مستوى الحزمة وكل متغير محلي لكل دالة فعالة حاليًا يمكن أن يكون بداية أو جذر لمسار للمتغير المشكوك فيه (المتغير الذي يمكن استرجاع حافظته) أو مؤشرًا أو غيره من المراجع التي قد تقودنا في النهاية إلى المتغير. في حال عدم وجود مثل هذا المسار فإنه يتعذر الوصول للمتغير وبالتالي لن يؤثر المتغير على بقية الحسابات.

وكون عمر المتغير يحدد حسب إمكانية الوصول إليه، فالمتغير المحلي قد يصمد لتكرار واحد من الحلقة المضمنة، وقد يستمر بقاؤه حتى بعد رجوع دالته المضمنة.

وقد يقوم المترجم (المحول البرمجي) بوضع المتغيرات المحلية في الذاكرة التكويمية (heap) أو المكدس (stack) (تعني المكدس أنه كدس بعضه فوق بعض)، علما أن هذا الخيار لا يعتمد على استخدام var أو new في إعلان المتغير.

```
var global *int
func f() {
     var x int
     x = 1
     global = &x
}

func g() {
y := new(int)
*y = 1
}
```

لاحظ هنا أنه لا بد من وضع x في الذاكرة التكويمية لأنه ما زال ممكنًا الوصول إليه من المتغير global بعد إعادة f بالرغم من إعلانه كمتغير محلي، وبالتالي نقول إن x تهرب من f، وعلى النقيض من ذلك عندما تعود g فإن المتغير y* يصبح متعذر الوصول إليه ويمكن إعادة استحدامه. وبما أن y* لا تهرب من g فمن الآمن على المترجم أن يضع y* في المكدس بالرغم من تخصيصه باستخدام new، على أية حال لا داعي للقلق بشأن فكرة الهروب عند كتابة الكود ولكن من الجيد أن تضع في ذهنك عند تحسين الأداء كون كل متغير يهرب فإنه يتطلب مواقع إضافية في الذاكرة.

يقدم جامع النفايات مساعدة ضخمة في كتابة البرامج الصحيحة ولكنه لن يريحك من عبء التفكير في الذاكرة، فأنت لست بحاجة لتحديد موقع الذاكرة وتفريغها بشكل مباشر ولكن من أجل كتابة برامج فعالة أنت بحاجة إلى معرفة عمر المتغيرات. على سبيل المثال عندما تحتفظ بالمؤشرات للعناصر قصيرة العمر ضمن العناصر طويلة العمر وخصوصًا في المتغيرات العالمية (غير المحلية) فإنك ستمنع جامع النفايات من إعادة تدوير العناصر قصيرة العمر.

2.4 التعيين Assignments

تحديث قيمة المتغير من خلال جملة التعيين، وأبسط صورة لجملة التعيين هي وجود متغير على الجهة اليسرى من إشارة المساواة = ووجود تعبير على اليمين.

```
x = 1                    //named variable
*p = true                //indirect variable
person.name = "bob"      //struct field
count[x] = count[x] * scale //array or slice or map element
```

جميع العمليات الحسابية والمعالجة الثنائية (binary) تحتوي على عامل التعيين الذي بدوره على سبيل المثال جعل الجملة الأخيرة تكتب كالتالي:

```
count[x] *= scale
```

مما يوفر علينا جهد تكرار وإعادة تقييم التعابير للمتغيرات.

يمكن زيادة أو تنقيص المتغيرات الرقمية باستخدام ++ و --:

```
v := 1
v++  // same as v = v + 1; v becomes 2
v--  // same as v = v - 1; v becomes 1 again
```

2.4.1 تعين مجموعة Tuple Assignment

يوجد شكل آخر للتعيين وهو تعيين المجموعة، حيث يقوم بتعيين مجموعة من المتغيرات مرة واحدة، بحيث تقيم جميع التعابير على الجهة اليمين قبل تحديث أي من المتغيرات، وتكمن فائدة هذه الجملة عندما يظهر المتغير في كلتا جانبي الجملة، على سبيل المثال عند مبادلة قيمة متغيرين مع بعضهما:

```
x, y = y, x
a[i], a[j] = a[j], a[i]
```

أو عند حساب القاسم المشترك الأكبر (GCD) لعددين صحيحين:

```
func gcd(x, y int) int {
    for y != 0 {
        x, y = y, x%y
    }
    return x
}
```

أو عند حساب أرقام فيبوناتشي (n-th Fibonacci) بشكل دوري:

```
func fib(n int) int {
    x, y := 0, 1
    for i := 0; i < n; i++ {
        x, y = y, x+y
    }
    return x
}
```

وتعين جملة تعيين المجموعة على جعل سلسلة من جمل التعيين السهلة أكثر ترتيبا:

```
i, j, k = 2, 3, 5
```

برغم أنه من الناحية الأسلوبية، تجنب استخدام تعيين المجموعة في حال كانت التعابير معقدة، فالجمل المنفصلة أسهل للقراءة من جملة واحدة معقدة.

بعض التعابير مثل استدعاء دالة ذات نتائج متعددة ستزودنا بالعديد من القيم وبالتالي عند استخدام مثل هذه الاستدعاءات يجب أن يوجد في الجانب الأيسر عدد من المتغيرات يساوي عدد النتائج الموجودة في الجانب الأيمن.

```
f, err = os.Open("foo.txt")          // function call returns two values
```

غالبًا ما تستخدم الوظائف تلك النتائج الإضافية لتحديد بعض الأخطاء من خلال إرجاع error كما هو الحال في استدعاء os.Open أو bool وغالبًا تستدعي ok، كما سنرى في الفصول القادمة ثلاثة عوامل تعمل بنفس المبدأ. في حال تواجد بحث الخريطة map lookup (راجع القسم 4.3) أو توكيد النمط (type assertion) (راجع القسم 7.10) أو

استقبال القناة (channel receive) (راجع القسم 8.4.2) في جملة تكون كلا نتائجها متوقعة فإن كل منهما ينتج نتائج منطقية إضافية:

```
v, ok = m[key]        // map lookup
v, ok = x.(T)         // type assertion
v, ok = <-ch          // channel receive
```

كما هو الحال في إعلان المتغيرات يمكننا تعيين قيم غير مطلوبة للمعرّف الفارغ:

```
_, err = io.Copy(dst, src)  // discard byte count
_, ok = x.(T)              // check type but discard result
```

2.4.2 قابلية التعيين Assignability

جملة التعيين هي عبارة عن صورة مباشرة للتعيين بشكل عام، ولكن هناك مواضع عديدة في البرنامج حيث يحدث التعيين بشكل ضمني، على سبيل المثال يقوم استدعاء دالة بتعيين قيمة وسيطة لمعاملات المتغيرات بشكل ضمني، مثال آخر تقوم جملة return ضمنيا بتعيين معاملات الرجوع للمتغيرات الناتجة، ومثال آخر التعبير literal للأنماط المركبة كما هو موضح في الشريحة أدناه (راجع القسم 4.2)

```
medals := []string{"gold", "silver", "bronze"}
```

يقوم ضمنيا بتعيين كل عنصر كما لو أنه مكتوب بالصورة التالية:

```
medals[0] = "gold"
medals[1] = "silver"
medals[2] = "bronze"
```

وكذلك يحدث مع عناصر الخرائط والقنوات، برغم أنها ليست متغيرات عادية، إلا أنها تكون قابلة للتعيين الضمني.

تكون جملة التعيين صحيحة سواء أكانت مباشرة أم غير مباشرة طالما الجزء الأيسر (المتغيرات) والجزء الأيمن (القيم) لديهم نفس النوع، وبطرقة أكثر عمومية، إن جملة التعيين صحيحة طالما القيمة قابلة للتعيين طبقًا لنوع المتغير.

يوجد لقاعدة قابلية التعيين عدة حالات، وبالتالي سنشرح كل حالة عند شرح كل نوع، بالنسبة للأنواع التي شرحناها حتى الآن لها نفس القواعد وهي أن النوع يجب أن يتشابه تماما ويمكن تعيين قيمة nil لأي نوع واجهة أو نوع مرجعي. الثوابت (راجع القسم 3.6) لها قواعد أكثر مرونة من جهة قابلية التعيين من أجل تجنب الحاجة لأكثر التحويلات المباشرة.

سواء تمت مقارنة أي قيمتين من خلال == أو != وانطبق عليهما قواعد قابلية التعيين فإن المعامل الأول قابل للتعيين في المعامل الثاني والعكس صحيح. وبالإضافة إلى قابلية التعيين سنشرح شيء إضافي يسمى قابلية المقارنة (comparability) عند شرح كل نوع جديد.

2.5 إعلان النمط Type Declarations

يحدد نوع المتغير أو التعبير خصائص القيمة التي قد تخزن فيه، مثل الحجم (عدد البت أو عدد العناصر) وكيفية تمثيلهم داخليًا والعمليات الأساسية الممكن إجراؤها عليهم والطرق المرتبطة بهم.

في أي برنامج يوجد متغيرات لها نفس التمثيل ولكن تدل على مفهوم مختلف تماما، على سبيل المثال يمكن استخدام int لتمثيل فهرس التكرار أو طابع زمني أو واصف ملف أو شهر من أشهر السنة، ويمكن استخدام float64 لتمثيل السرعة بالمتر لكل ثانية أو درجة الحرارة حسب مقياس معين، ومن الممكن استخدام string لتمثيل كلمة مرور أو اسم لون.

إعلان النوع يحدد نوع جديد مسمى له نفس النوع الأساسي للنوع الموجود، يساعدنا النوع المسمى على الفصل بين الاستخدامات المختلفة للنوع الأساسي مما يقلل احتمالية الخلط بينهما.

```
type name underlying-type
```

غالبا ما يكون إعلان النوع على مستوى الحزمة بحيث يكون النوع المسمى ظاهرا في كامل الحزمة، وفي حال كان الاسم صادرا (أي يبدأ بحرف كبير) فإنه يمكن الوصول إليه من خلال حزم أخرى.

لتوضيح إعلان النوع سنقوم بتحويل موازين الحرارة إلى أنواع مختلفة:

```
gopl.io/ch2/tempconv0
// Package tempconv performs Celsius and Fahrenheit temperature computations.
package tempconv
import "fmt"
type Celsius float64
type Fahrenheit float64
const (
    AbsoluteZeroC = -273.15
    FreezingC     = 0
    BoilingC      = 100
)
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

كما تلاحظ أن هذه الحزمة تحدد النوعين (مئوية وفهرنهايت) كوحدتين لقياس الحرارة، مع ملاحظة أن كلاهما نفس النمط الأساسي float64 ولكنهم ليس نفس النوع، وبالتالي لا يمكن مقارنتهما أو جمعهما بالعمليات الحسابية، عند المقدرة

على التمييز بين الأنواع ستتمكن من تجنب الأخطاء مثل جمع درجتين حرارة من مقياسين مختلفين، وبالتالي هناك الحاجة لنمط تحويل مباشر مثل Celsius(t) و Fahrenheit(t) للتحويل من float64. أما Celsius(t) و Fahrenheit(t) فهما تحويلات وليس استدعاءات دوال. فعليًا لن يتم تغيير القيمة أو التمثيل إنما سيغير المعنى المباشر، من ناحية أخرى تقوم الدالتين CToF و FToC بالتحويل بين المقياسين مع إرجاع قيم مختلفة.

لكل نوع (على سبيل المثال النوع T) يقابلها عملية تحويل T(x) حيث تقوم بتحويل القيمة x إلى النوع T، من الممكن التحويل من نوع للآخر في حال كانا لهما نفس النوع الأساسي أو كانا كلاهما نوع مؤشر غير مسمى يشيرا إلى متغيرين من نفس النوع الأساسي، فهذه التحويلات تغير النوع ولكن لا تغير تمثيل القيمة، في حال كانت x قابلة للتعيين كقيمة لـ T فالتحويل يكون ممكنا ولكن لا داعي له.

ومن الممكن إجراء التحويلات بين الأنواع الرقمية وممكنا بين أنماط السلاسل والشرائح كما هو موضح في الفصول اللاحقة. تلك التحويلات قد تغير تمثيل القيمة، على سبيل المثال تحويل عدد النقطة العائمة (floating-point) إلى عدد صحيح (integer) سيقوم بإهمال الجزء العشري للرقم، وتحويل سلسلة نصية إلى شريحة [ ]byte سيصنع نسخة من بيانات السلسلة، وفي جميع الحالات لن يفشل التحويل خلال تنفيذ البرنامج.

تحدد الأنواع الأساسية بنية وتمثيل النوع المسمى والعمليات الأساسية التي يدعمها وهي بالعادة مشابهة لعمليات النوع الأساسي في حال استخدامها مباشرة، وذلك يعني أن العمليات الحسابية الممكن إجراؤها على Celsius و Fahrenheit هي نفس العمليات الممكن إجراؤها على float64.

```
fmt.Printf("%g\n", BoilingC-FreezingC) // "100" °C
boilingF := CToF(BoilingC)
fmt.Printf("%g\n", boilingF-CToF(FreezingC)) // "180" °F
fmt.Printf("%g\n", boilingF-FreezingC)         // compile error: type
mismatch
```

ومن الممكن استخدام عمليات المقارنة مثل == و> لمقارنة قيمة النوع المسمى مع قيمة أخرى من نفس النوع المسمى أو النوع الأساسي، ولكن لا يمكن مقارنة قيمتين من نوعين مسميين مختلفين بشكل مباشر:

```
var c Celsius
var f Fahrenheit
fmt.Println(c == 0)         // "true"
fmt.Println(f >= 0)        // "true"
fmt.Println(c == f)        // compile error: type mismatch
fmt.Println(c == Celsius(f)) // "true"!
```

لاحظ في الحالة الأخيرة بالرغم من الاسم فإن تحويل النمط Celsius(f) لم يغير قيمة معامله إنما غير نوعه فقط، وفحص الجملة يعطي أنها جملة صحيحة لأن كلا c و f يساوي صفر.

يمنحنا النوع المسمى تسهيلات تساعدنا في تجنب كتابة أنواع معقدة مرارًا وتكرارًا، وتكون الفائدة قليلة عندما يكون النوع الأساسي بسيطًا مثل float64 ولكن تكون الفائدة كبيرة عندما تكون الأنواع معقدة كما سنشاهد عند شرح السلاسل.

وتمكننا الأنواع المسماة من تعريف سلوكيات جديدة لقيم النوع، توصف تلك السلوكيات كمجموعة من الوظائف المرتبطة بالنوع وتسمى طرق النوع (type's method)، سنشرح الطرق بالتفصيل في الفصل السادس ولكن سنتطرق إليه بشكل عام هنا.

لاحظ الإعلان أدناه حيث معامل c Celsius موجود قبل اسم الدالة مرتبط بنوع Celsius دالة تسمى String تعيد القيمة العددية للمعامل c متبوعة بـ C°:

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

العديد من الأنواع تعلن أسلوب String بهذا الشكل لأنه يسهل التحكم بكيفية ظهور قيم النوع عند طباعتها كسلسلة بواسطة حزمة fmt كما سنرى في القسم 7.1.

```
c := FToC(212.0)
fmt.Println(c.String())    // "100°C"
fmt.Printf("%v\n", c)         // "100°C"; no need to call String explicitly
fmt.Printf("%s\n", c)         // "100°C"
fmt.Println(c)             // "100°C"
fmt.Printf("%g\n", c)         // "100"; does not call String
fmt.Println(float64(c))    // "100"; does not call String
```

2.6 الحزم والملفات Packages and Files

وظيفة الحزم في لغة جو هي نفس وظيفة المكتبات أو الوحدات البرمجية في لغات البرمجة الأخرى وتتمتع الحزم بقابلية التركيب والتغليف والتجميع المفصل وإعادة الاستخدام. توجد الشفرة المصدرية للحزم في ملف أو عدة ملفات (go.) وغالبًا في الدليل الذي ينتهي اسمه بمسار الاستيراد، على سبيل المثال ملفات حزمة (gopl.io/ch1/helloworld) محفوظة في الدليل ($GOPATH/src/gopl.io/ch1/helloworld).

تخدم كل حزمة كحيز اسم منفصل لكل الإعلانات الخاصة بها، على سبيل المثال يوجد ضمن حزمة image المعرف Decode ليشير إلى دالة مختلفة عما يشير إليه المحدد في حزمة unicode/utf16. للإشارة إلى دالة من خارج الحزمة يجب علينا تأهيل المحدد لجعله صريحا سواء نقصد image.Decode أو utf16.Decode.

تتيح لنا الحزم إمكانية إخفاء المعلومات من خلال التحكم بإمكانية رؤية الأسماء خارج الحزمة أو تصديرها، في لغة البرمجة جو توجد قاعدة بسيطة تحدد إذا ما كان المحددات مصدرة أو غير مصدرة ، وذلك من خلال أول حرف في الاسم، فإذا كان الحرف كبيرًا فالاسم مصدرًا.

لتوضيح هذه الأساسيات، افترض أن برنامجنا لتحويل درجات الحرارة أصبح مشهورا ونريد نشره بين مستخدمي لغة جو كحزمة جديدة، ماذا يجب علينا فعله؟

دعنا ننشئ حزمة تسمى gopl.io/ch2/tempconv وهي تنويعة من المثال السابق. (قمنا هنا بعمل استثناء في ترقيم الأمثلة ضمن الترتيب وذلك ليكون مسار الحزمة أكثر واقعية) ستكون الحزمة مخزنة في ملفين لتوضيح كيفية الوصول إلى إعلانات في ملفات منفصلة من الحزمة، في الواقع العملي نستخدم ملف واحد فقط لحزمة صغيرة مثل هذه.

قمنا بإعلان الأنماط والثوابت والطرق في (tempconv.go):

```
gopl.io/ch2/tempconv
// Package tempconv performs Celsius and Fahrenheit conversions.
package tempconv
import "fmt"
type Celsius float64
type Fahrenheit float64
const (
    AbsoluteZeroC = -273.15
    FreezingC     = 0
    BoilingC      = 100
)
func (c Celsius) String() string    { return fmt.Sprintf("%g°C", c) }
func (f Fahrenheit) String() string { return fmt.Sprintf("%g°F", f) }
```

ودوال التحويل في (conv.go):

```
package tempconv
// CToF converts a Celsius temperature to Fahrenheit.
func CToF(c Celsius) Fahrenheit { return Fahrenheit(c*9/5 + 32) }
// FToC converts a Fahrenheit temperature to Celsius.
func FToC(f Fahrenheit) Celsius { return Celsius((f - 32) * 5 / 9) }
```

يبدأ كل ملف بإعلان الحزمة التي تحدد اسم الحزمة، عند تصدير الحزمة يشار إلى عناصرها tempconv.CToF وهلم جرا. الأسماء على مستوى الحزمة مثل الأنماط والثوابت المعلن عنها في ملف في الحزمة متاحة لجميع الملفات الأخرى لنفس الحزمة كما لو أن كل الشفرة المصدرية في ملف واحد، لاحظ أن tempconv.go يستورد fmt ولكن conv.go لا يفعل ذلك لأنه لا يستخدم أي شيء من fmt.

وكون أسماء الثوابت على مستوى الحزمة تبدأ بحرف كبير فيمكن الوصول إليها بأسماء مؤهلة مثل tempconv.AbsoluteZeroC:

```
fmt.Printf("Brrrr! %v\n", tempconv.AbsoluteZeroC) // "Brrrr! -273.15°C"
```

لتحويل درجة الحرارة من مئوية إلى فهرنهايت ضمن حزمة تستورد gopl.io/ch2/tempconv يمكننا أن نكتب الكود التالي:

```
fmt.Println(tempconv.CToF(tempconv.BoilingC)) // "212°F"
```

التعليق (doc comment) (راجع القسم 10.7.4) يتقدم فورا على إعلان الحزمة لتوثيق الحزمة ككل. وتوافقيا، يجب أن تبدأ بجملة تلخيصيه حسب الأسلوب الموضح. يجب أن يوجد تعليق توثيق الحزمة (doc comment) في ملف واحد فقط من الحزمة، أما تعليقات التوثيق الموسعة عادة ما توجد في ملف خاص بها يسمى doc.go.

تمرين 2.1: قم بإضافة أنواع وثوابت ووظائف إلى tempconv لمعالجة الحرارة حسب ميزان كلفين بحيث أن الصفر في مقياس كلفين يساوي 273.15- درجة مئوية وفارق ألف له نفس قيمة درجة واحدة مئوية.

2.6.1 الاستيراد import

تُعرَّف كل حزمة في برمجية جو بسلسلة نصية فريدة تسمى مسار الاستيراد (import path) حيث أن تلك السلاسل تظهر في إعلان الاستيراد مثل (gopl.io/ch2/tempconv). خصائص اللغة لا تحدد مصدر هذه السلاسل ولا توضح معناها ولكن تقوم الأدوات بترجمتها، عند استخدام أدوات جو (راجع الفصل العاشر) سيزودنا مسار الاستيراد بدليل يحتوي على واحد أو أكثر من ملفات المصدر جو التي تشكل مجتمعة الحزمة.

بالإضافة إلى مسار الاستيراد، كل حزمة تمتلك اسما قصيرا (ليس شرطا أن يكون فريدا) يظهر في إعلان الحزمة. وبالاتفاق، يتطابق اسم الحزمة مع آخر قسم من مسار الاستيراد، مما يجعل من السهولة توقع اسم الحزمة لمسار الاستيراد (gopl.io/ch2/tempconv هو tempconv.

لاستخدام حزمة (gopl.io/ch2/tempconv يجب أن نستوردها:

```
gopl.io/ch2/cf
// Cf converts its numeric argument to Celsius and Fahrenheit.
package main
import (
    "fmt"
    "os"
    "strconv"
    "gopl.io/ch2/tempconv"
)
```

```
func main() {
    for _, arg := range os.Args[1:] {
        t, err := strconv.ParseFloat(arg, 64)
        if err != nil {
            fmt.Fprintf(os.Stderr, "cf: %v\n", err)
            os.Exit(1)
        }
        f := tempconv.Fahrenheit(t)
        c := tempconv.Celsius(t)
        fmt.Printf("%s = %s, %s = %s\n",
            f, tempconv.FToC(f), c, tempconv.CToF(c))
    }
}
```

يربط إعلان الاستيراد اسما قصيرا مع الحزمة المستوردة لاستخدامه للإشارة إلى محتويات الحزمة في الملف بالكامل. الاستيراد أعلاه يجعلنا نشير للأسماء ضمن gopl.io/ch2/tempconv من خلال استخدام معرف مؤهل مثل tempconv.CToF، في الوضع الافتراضي يكون الاسم القصير هو نفس اسم الحزمة (في مثالنا الاسم هو tempconv) ولكن إعلان الاستيراد قد يحدد اسمًا بديلا لتجنب التعارض (راجع القسم 10.3).

يقوم البرنامج cf بتحويل معامل سطر الأوامر الرقمي إلى قيمته في كلا المقياسين المئوي وفهرنهايتي:

```
$ go build gopl.io/ch2/cf
$ ./cf 32
32°F = 0°C, 32°C = 89.6°F
$ ./cf 212
212°F = 100°C, 212°C = 413.6°F
$ ./cf -40
-40°F = -40°C, -40°C = -40°F
```

من الخاطئ استيراد الحزمة ومن ثم عدم الإشارة إليها، فهذا الفحص يساعد على منع التبعيات (الارتباطات) التي قد تصبح غير ضرورية مع تطور الكود ومن الممكن أن تكون مزعجة عند فحص البرنامج لتصحيح الأخطاء، وبما أن إزالة تعليق سطر من الكود مثل log.Print("got here!") قد يسبب في إزالة المرجعية الوحيدة لاسم الحزمة log مما يسبب في تشويش المترجم ليعتبرها خطأ، في هذه الحالة يجب عليك وضع علامة التعليق أو مسح الاستيرادات غير الضرورية.

ومن المفضل استخدام الأداة (golang.org/x/tools/cmd/goimports) حيث تقوم تلقائيا بإدراج وحذف الحزم من الإعلان المستورد حسب الحاجة، معظم برامج التحرير يمكن أن تعد لتشغيل goimports في كل مرة تحفظ فيها الملف، ومثل الأداة gofmt، فإنها تقوم بتهيئة ملف الشفرة البرمجية ليصبح جميل المظهر وبصيغة متعارف عليها.

تمرين 2.2: اكتب برنامجا تحويل الوحدات متعدد الاستخدامات يشبه cf بحيث يقرأ الأرقام من معامل سطر الأوامر أو من المدخل القياسي في حال عدم وجود معاملات، وقم بتحويل كل رقم إلى وحدة مثل درجة الحرارة بالمئوية وفهرنهايت والطول بوحدة القدم والمتر والوزن بوحدة الباوند والكيلوغرام وهلم جرا.

2.6.2 تهيئة الحزمة Package Initialization

أول خطوة في تهيئة الحزمة هي تهيئة المتغيرات على مستوى الحزمة حسب ترتيب إعلاناتهم ما عدا الاعتماديات حيث تحل أولا:

```
var a = b + c   // a initialized third, to 3
var b = f()     // b initialized second, to 2, by calling f
var c = 1       // c initialized first, to 1
func f() int { return c + 1 }
```

في حال احتواء الحزمة على أكثر من ملف (.go) فيتم تهيئتهم حسب الترتيب الذي تمرر للمترجم، تقوم أداة (go) بترتيب ملفات (.go) حسب الاسم قبل استحضار المترجم.

يبدأ عمر كل متغير معلن على مستوى الحزمة بقيمة تعبير التهيئة الخاص به، بعض المتغيرات مثل جداول البيانات سيكون تعبير التهيئة الخاص به معقد جدًا وبالتالي نستخدم الدالة init كونها أبسط، أي ملف قد يحتوي على أي رقم لوظائف تكون إعلاناتها كالتالي:

```
func init() { /* ... */ }
```

لا يمكن استدعاء وظائف مثل init أو جعل لها مرجعية، أما غيرها من الوظائف يمكن استدعاؤها وجعل مرجعية لها. تنفذ وظائف init تلقائيا ضمن أي ملف عند بدأ البرنامج حسب ترتيب إعلاناتهم.

تهيئ كل حزمة على حدة على حسب ترتيب استيرادها في البرنامج، والاعتماديات أولا(هي الحزم التي تعتمد على غيرها ، فمثلا لو الحزمة p تستورد الحزمة q فإن الحزمة q يتم تهيئتها كاملة قبل البدء بتهيئة الحزمة p، وتكون التهيئة على حسب التصميم من أسفل لأعلى، وبالتالي حزمة main تهيء آخر حزمة، أي تهيء جميع الحزم قبل البدء بتهيئة الحزمة main.

تحدد الحزمة أدناه الدالة PopCount حيث ترجع رقما مكونا من مجموعة من البت bits، بحيث توجد البت التي قيمتها 1 في قيمة uint64 وتسمى "تعداد السكان population count"، وتستخدم الدالة init للتحضير المسبق لجدول النتائج و pc لكل قيمة 8 بت محتملة وبالتالي لا تحتاج الدالة PopCount إلى إجراء 64 خطوة ولكن بإمكانها إرجاع مجموع 8

عمليات تنقيب في الجدول (بكل تأكيد هذه ليست أسرع خوارزمية لعد البت ولكنها مناسبة لفهم مبدأ الدالة init وتوضيح كيفية الحساب المسبق لجدول القيم كونه تقنية برمجة مفيدة).

```
gopl.io/ch2/popcount
package popcount
// pc[i] is the population count of i.
var pc [256]byte
func init() {
    for i := range pc {
        pc[i] = pc[i/2] + byte(i&1)
    }
}
// PopCount returns the population count (number of set bits) of x.
func PopCount(x uint64) int {
    return int(pc[byte(x>>(0*8))] +
        pc[byte(x>>(1*8))] +
        pc[byte(x>>(2*8))] +
        pc[byte(x>>(3*8))] +
        pc[byte(x>>(4*8))] +
        pc[byte(x>>(5*8))] +
        pc[byte(x>>(6*8))] +
        pc[byte(x>>(7*8))])
}
```

لاحظ أن حلقة range في init تستخدم معاملًا واحدًا index مما يعني أن القيمة غير مهمة ولا يوجد داعي لوضعها، ومن الممكن كتابة الحلقة بالشكل التالي:

```
for i, _ := range pc {
```

سنرى استخدامات أخرى للدالة init في القسم التالي وفي القسم 10.5

تمرين 2.3: أعد كتابة PopCount ولكن استخدم الحلقة بدلًا من التعبير المنفرد، قارن أداء التطبيقين. (انظر للقسم 11.4 لمعرفة كيفية المقارنة بين تطبيقين بشكل منهجي).

تمرين 2.4: اكتب إصدارا آخر من PopCount بحيث يحصي البت من خلال نقل معاملاته عبر 64 موقع للبت وافحص البت الموجود أقصى اليمين في كل مرة وقارن نتائجه مع نسخة تفقد الجدول المذكور سابقًا.

تمرين 2.5: يقوم التعبير (x&(x-1 بمسح بت x الموجود أقصى اليمين في حال لم تكن قيمته صفر، اكتب إصدار من PopCount بحيث يحصي البت من خلال استخدام المعلومة المعطاة هذه وقيم أداءه.

2.7 النطاق Scope

يقوم الإعلان بربط الاسم مع كيان البرنامج مثل الوظائف والمتغيرات، ونطاق الإعلان هو جزء من الشفرة المصدرية حيث استخدام الاسم المعلن يشير إلى ذلك الإعلان.

لا تخلط بين النطاق والعمر (فترة البقاء)، فنطاق الإعلان هو المنطقة التي يوجد فيها نص البرنامج فهي متعلقة بوقت الترجمة، بينما فترة بقاء المتغير هي فترة زمنية ضمن التنفيذ حيث فيها يمكن الإشارة إلى المتغير من قبل أجزاء أخرى من البرنامج فهي متعلقة بوقت التنفيذ.

الكتلة النحوية (syntactic block) هي سلسلة من الجمل الموضوعة بين قوسين مثل تلك الأقواس المحيطة بنص الدالة أو الحلقة، فالاسم المعلن داخل الكتلة النحوية لن يظهر خارج الكتلة، فالكتلة تحصر إعلانه وتحدد نطاقه، يمكن تعميم فكرة الكتل لتشمل مجموعات أخرى من الإعلانات غير المحصورة بأقواس في كود المصدر، وبالتالي سندعوهم الكتل معجمية (lexical blocks)، وتوجد كتلة معجمية تشمل كامل الشفرة المصدرية وتسمى الكتلة الشاملة (universe block) وكتلة لكل حزمة، ولكل ملف، ولكل جمل for و if و switch و select و بكل تأكيد لكل كتلة نحوية صريحة.

إن الإعلان عن الكتلة النحوية يحدد نطاق الإعلان والذي قد يكون كبيرًا أو صغيرًا. فإعلانات الأنواع المبنية والدوال و الثوابت مثل int و len و true تكون في الكتلة الشاملة ويمكن الإشارة إليها خلال من جميع أنحاء البرنامج. ويمكن الإشارة إلى الإعلانات الموجودة خارج أي دالة على مستوى الحزمة من أي ملف في نفس الحزمة. الحزم المستوردة مثل حزمة fmt الموجودة في مثال tempconv قد تم إعلانها على مستوى الملف وبالتالي يمكن الإشارة إليها من نفس الملف ولكن لا يجوز الإشارة إليها من ملف آخر داخل نفس الحزمة دون إجراء استيراد آخر. وهناك العديد من الإعلانات المحلية مثل إعلان المتغير c في دالة tempconv.CToF فيمكن الإشارة إليها فقط من نفس الدالة أو ربما جزء من الدالة.

نطاق تسمية التحكم بالتدفق control-flow كما هي مستخدمة في جمل break و continue و goto هو عبارة عن دالة تضمين كاملة.

قد يحتوي البرنامج على إعلانات متعددة لنفس الاسم طالما كل إعلان موجود في كتلة نحوية مختلفة، على سبيل المثال يمكنك حجز متغير محلي له نفس اسم متغيرات أخرى في نفس مستوى الحزمة أو كما هو موضح في القسم 2.3.3 ستجد أنه بإمكانك إعلان معامل دالة يسمى new رغم أن دالة بهذا الاسم معلنة مسبقًا في الكتلة الشاملة، فلا تحاول القيام بالأمر كثيرًا لأنه كلما كبر نطاق إعادة الإعلان كلما ستشوش قارئ البرنامج.

عندما يواجه المترجم مرجعا لاسم فإنه يبحث عن الإعلان بدءا بالكتلة اللغوية التي تحتويه ثم الذي يليه وهكذا حتى يصل إلى الكتلة الشاملة، ففي حال لم يجد المترجم أي إعلان فإنه يبلغك برسالة خطأ مفادها أنه يوجد اسم غير معلن، ففي حال الإعلان عن اسم ضمن الكتلة الداخلية والكتلة الخارجية سيقوم المترجم بإيجاد الإعلان الموجود في الكتلة الداخلية أولًا، وبالتالي سيقوم الإعلان الداخلي بإخفاء الإعلان الخارجي مما يجعله غير قابل الوصول:

```go
func f() {}
var g = "g"
func main() {
    f := "f"
    fmt.Println(f) // "f"; local var f shadows package-level func f
    fmt.Println(g) // "g"; package-level var
    fmt.Println(h) // compile error: undefined: h
}
```

قد يوجد ضمن الدالة بعض الكتل النحوية المتداخلة بشكل عشوائي وبالتالي قد يغطي أحد الإعلانات الداخلية على إعلانات أخرى، فمعظم الكتل متشكلة من بنيات ضبط التدفق مثل جملة if وحلقة for، فالبرنامج أدناه فيه ثلاثة متغيرات مختلفة اسمها x لأن كل إعلان موجود في كتلة نحوية مختلفة. (هذا المثال يوضح قواعد النطاق علمًا أنه مكتوب بطريقة سيئة!)

```go
func main() {
    x := "hello!"
    for i := 0; i < len(x); i++ {
        x := x[i]
        if x != '!' {
            x := x + 'A' - 'a'
            fmt.Printf("%c", x) // "HELLO" (one letter per iteration)
        }
    }
}
```

تلاحظ أن التعبير x[i] و x + 'A' - 'a' كل منهما يشير إلى إعلان x من كتل مختلفة العمق، سنشرح هذا بعد قليل. (لاحظ أن التعبير الأخير لا يكافئ (unicode.ToUpper)

كما هو مذكور أعلاه ليست جميع الكتل النحوية تمثل سلسلة تعابير محاطة بالأقواس، فبعضها بالكاد محصورًا، عند النظر في حلقة for بالأعلى سنجد أنها تنشئ كتلتين نحويتين اثنتين، أحدهما يحجز نص الحلقة كاملًا والآخر ضمني يحجز المتغيرات المعلنة بشروط التهيئة مثل i، فنطاق المتغير المعلن ضمن الحجز الضمني الداخلي هو الشرط وجملة i++ ونص جملة for.

المثال أدناه يوجد فيه أيضًا ثلاثة متغيرات اسمها x وكل منها قد تم إعلانه في كتلة مختلفة (أحدها في جسم نص الدالة وآخر في كتلة جملة for وآخر في كتلة الحلقة) ولكن فقط كتلتين منهما واضحتان:

```go
func main() {
    x := "hello"
    for _, x := range x {
        x := x + 'A' - 'a'
        fmt.Printf("%c", x) // "HELLO" (one letter per iteration)
    }
}
```

كما هو الحال في حلقة for فإن جملتي if و switch تنشئا أيضًا كتل مضمنة غير كتل نصيهما، الكود التالي في سلسلة if-else يوضح مجال x و y:

```
if x := f(); x == 0 {
    fmt.Println(x)
} else if y := g(x); x == y {
    fmt.Println(x, y)
} else {
    fmt.Println(x, y)
}
fmt.Println(x, y) // compile error: x and y are not visible here
```

جملة if الثانية متداخلة مع جملة if الأولى وبالتالي المتغيرات المعلنة ضمن مهيئ الجملة لأولى مرئية للجملة الثانية، وكذلك يكون الوضع مع جملة switch حيث توجد كتلة للشرط وكتلة لكل نص فرعي.

لا يوجد تأثير لترتيب الإعلانات على نطاقها على مستوى الحزمة، وبالتالي قد يشير الإعلان إلى نفسه أو إلى إعلان آخر يتبعها، مما يتيح لدينا إمكانية إعلان أنواع ووظائف متداخلة، سيبلغ المترجم عن رسالة خطأ في حال إشارة ثابت أو متغير إلى نفسه.

انظر إلى البرنامج أدناه:

```
if f, err := os.Open(fname); err != nil { // compile error: unused: f
    return err
}
f.ReadByte() // compile error: undefined f
f.Close()          // compile error: undefined f
```

نطاق f هو مجرد جملة if، وبالتالي لا يمكن لـ f الوصول للجملة التي تليها وبالتالي سيجد المترجم خطأ، في بعض أنواع المترجمات ستبلغ عن أخطاء إضافية تشير إلى أن المتغير f لم يستخدم مطلقا.

وبالتالي من الضروري الإعلان عن f قبل الشرط حتى يمكن الوصول إليه فيما بعد:

```
f, err := os.Open(fname)
if err != nil {
     return err
}
f.ReadByte()
f.Close()
```

قد ترغب بمحاولة تجنب إعلان f و err في الكتلة الخارجية من خلال تحريك الاستدعاءات إلى ReadByte و Close داخل حجز else:

```
if f, err := os.Open(fname); err != nil {
    return err
```

```
} else {
    // f and err are visible here too
    f.ReadByte()
    f.Close()
}
```

ولكن الوضع الطبيعي في لغة البرمجة جو هو التعامل مع الخطأ في كتلة if ومن ثم العودة، وبالتالي يسير مسار تنفيذ البرنامج بشكل ناجح.

تتطلب إعلانات المتغيرات القصيرة معرفة جيدة للنطاق، انظر للبرنامج أدناه، ستجده يبدأ بإيجاد دليل عمله الحالي ويحفظه في متغير على مستوى الحزمة، يمكن تنفيذ ذلك من خلال استدعاء os.Getwd في دالة main، ولكن من المفضل فصله عن المنطق الأساسي خصوصا في حالات الفشل في الحصول على الدليل بسبب خطأ فادح، تقوم الدالة log.Fatalf بطباعة رسالة واستدعاء os.Exit(1).

```
var cwd string
func init() {
    cwd, err := os.Getwd() // compile error: unused: cwd
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

كونه لم يعلن عن كلا cwd و err في كتلة الدالة init فإن جملة =: تعلنهما كلاهما كمتغيرات محلية، إن الإعلان الداخلي لـ cwd تجعل الكتلة الخارجية غير قابلة للوصول، وبالتالي الجملة لا تحدّث متغير cwd على مستوى الحزمة كما هو مطلوب.

تستطيع مترجمات لغة جو الحالية كشف أنه لم يستخدم المتغير cwd المحلية وسيبلغ عن ذلك كخطأ، ولكن المترجم غير مجبور على أداء هذا الفحص، بالإضافة إلى ذلك يمكن التغلب على فحص المترجم من خلال إجراء تغيير طفيف مثل إضافة جملة تنتج خرجا تشير إلى cwd المحلي.

```
var cwd string
func init() {
    cwd, err := os.Getwd() // NOTE: wrong!
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
    log.Printf("Working directory = %s", cwd)
}
```

تبقى متغيرات cwd غير المحلية غير مهيئة، ولكن من الواضح أن مخرج log يتجنب الخطأ.

هناك الكثير من الطرق للتعامل مع مثل هذه المشاكل ولكن أكثرها مباشرة هي تجنب =: من خلال تعريف err ضمن إعلان var منفصل:

```
var cwd string
func init() {
    var err error
    cwd, err = os.Getwd()
    if err != nil {
        log.Fatalf("os.Getwd failed: %v", err)
    }
}
```

لقد تعلمنا عن الحزم والملفات والإعلانات والجمل التي تصف بنية البرنامج، في الفصلين التاليين سنتعلم عن بنية البيانات.

3- أنواع البيانات الأساسية

إن البيانات كلها عبارة عن عدد من البِتات (bits) في النهاية، ولكن الكمبيوتر يعمل بشكل أساسي على عدد ثابت منها اسمه "كلمات" (words)، والتي يفسرها الكمبيوتر كأعداد صحيحة، أو أرقام فاصلة عائمة (floating-point)، أو مجموعات بيتات (bits)، أو عناوين ذاكرة، ثم يدمجها في مجموعات أكبر تمثل حِزم (packets) وبيكسلات (pixels)، وحافظات (portfolios)، وكل شيء آخر. تقدم لغة جو مجموعة طرق متنوعة لتنظيم البيانات، مع نطاق من أنواع البيانات التي تطابق خصائص العتاد (hardware) من ناحية، بينما تقدم من ناحية أخرى ما يطلبه المبرمجون لتمثيل بِنيات البيانات المعقدة بشكل مريح.

تنقسم أنواع جو إلى أربع فئات هي: الأنواع الأساسية (basic types) والأنواع المُجمعة (aggregate type)، والأنواع المرجعية (reference type)، وأنواع الواجهة (interface type). إن الأنواع الأساسية هي موضوع هذا الفصل، وتتضمن الأرقام والسلاسل النصية (strings) والقيم المنطقية (Booleans). وتتضمن الأنواع المجمعة كل من المصفوفات (arrays) (راجع 4.1) والبِنيات (structs) (راجع 4.4)، وهي تُشكّل أنواع بيانات أكثر تعقيدًا من خلال دمج قيم أنواع عديدة أبسط. تُعَد الأنواع المرجعية مجموعة متنوعة تتضمن المؤشرات (pointers) (راجع 2.3.2)، والشرائح (slices) (راجع 4.2)، والخرائط (maps) (راجع 4.3)، والوظائف (functions) (الفصل الخامس)، والقنوات (channels) (الفصل الثامن)، ولكن ما يجمع بينها هو أنها كلها تشير إلى متغيرات البرنامج أو حالته بشكل غير مباشر، بحيث أن تأثير أي عملية على مرجع واحد يُلاحظ في كل نُسخ هذا المرجع. أخيرًا، سنتحدث عن أنواع الواجهة في الفصل السابع.

3.1 الأعداد الصحيحة (integers)

تتضمن أنواع بيانات جو الرقمية أعداد صحيحة ذات أحجام متعددة، وأرقام فاصلة عائمة (floating point)، وأرقام معقدة. يحدد كل نوع رقمي حجم وتوقيع قيمه. لنبدأ بالأعداد الصحيحة.

تقدّم جو حساب أعداد صحيحة موقّعة وغير موقّعة. وهناك أربع أحجام مختلفة من الأعداد الصحيحة الموقّعة – وهي 8، و 16، و 32، و 64 بت – ويمثلها الأنواع int8 و int16 و Int32 و int64، والنسخة غير المُوقّعة المناظرة لها هي uint8 و uint16 و uint32 و uint64.

يوجد أيضًا نوعين يُطلق عليهما int و uint فقط، وهما محايدين، أو يمثلا أكثر الأحجام كفاءة للأعداد الصحيحة الموقعة وغير الموقعة في منصة معينة. تُعد int النوع الرقمي الأكثر استخدامًا، وتُستخدم على نطاق واسع. إن هذين النوعين حجمها واحد، سواء أكان 32 أم 64 بت، ولكن لا يجب على المرء افتراض أي افتراضات حول أيهما، لأن المترجمات المختلفة يمكن أن تتخذ خيارات مختلفة حتى في العتاد المتطابق.

إن النوع rune هو مرادف لـ int32، ويشير عادة إلى أن القيمة هي نقطة شفرة Unicode. يمكن استخدام الاسمين بالترادف. وبالمثل، النوع byte مرادف لـ uint8، ويؤكد على أن القيمة عبارة عن قطعة من البيانات الخامة وليست كمية رقمية صغيرة.

أخيرًا، هناك نوع العدد الصحيح غير الموقّع uintptr، الذي لم يُحدد عرضه، ولكنه كاف لحمل كل بِتات قيمة المؤشر. يُستخدم النوع uintptr في البرمجة منخفضة المستوى، كبرمجة على حدود برنامج جو مع مكتبة C أو نظام تشغيل. سنرى أمثلة على هذا عندما نتعامل مع حزمة unsafe في الفصل 13.

بغض النظر عن أحجامهم، تعتبر int و uint و uintptr أنواع مختلفة عن أشقائها ذوي الأحجام الصريحة. من ثم، فإن int ليست نفس نوع int32، حتى لو كان الحجم الطبيعي للأعداد الصحيحة هو 32 بِت، ويتطلب تحويلا صريحا لاستخدام قيمة int عندما تكون int32 مطلوبة، والعكس صحيح.

تُمثّل الأرقام الموقعة في حاويتين، حيث يُحفَظ البِت الأعلى لعلامة الرقم، و الثاني لنطاق قيم الرقم حيث عدد n بت هو من -2^{n-1} إلى $2^{n-1}-1$. وتستخدم الأعداد الصحيحة غير الموقعة كامل البِتات للقيم غير السالبة، وبالتالي يكون نطاقها من صفر إلى $2^{n}-1$. على سبيل المثل، نطاق int8 هو -128 إلى 127، بينما أن نطاق uint8 هو صفر إلى 255.

نستعرض هنا معاملات جو الثنائية الحسابية والمنطقية و المقارنة مرتبة على حسب الأولية:

```
* / % << >> & &^
+ - | ^
== != < <= > >=
&&
||
```

هناك خمس مستويات أسبقية (precedence) فقط للعوامل الثنائية (binary operators). إن العوامل التي على نفس المستوى مرتبطة باليسار، وبالتالي قد تكون الأقواس مطلوبة لمزيد من التوضيح أو لجعل العوامل تقيّم بالترتيب المستهدف بتعبير مثل هذا: mask & (1 << 28).

إن كل عامل في أول سطرين في الجدول أعلاه، + كمثال، له عامل تعيين مناظر له مثل += يمكن استخدامه لاختصار بيان تعيين.

إن العوامل الحسابية للأعداد الصحيحة مثل + و - و * و / يمكن تطبيقها على العدد الصحيح، والفاصلة العائمة، والأرقام المعقدة، ولكن عامل الباقي % ينطبق على الأعداد الصحيحة فقط. إن سلوك عامل الباقي% مع الأرقام السالبة يتفاوت بين لغات البرمجة. وفي لغة Go، تكون علامة الباقي دائمًا نفس علامة المقسوم، بالتالي 3%5- و 5%-3- كلاهما يساوي -2. يعتمد سلوك العامل(/) على ما إذا كان عواملها أعداد صحيحة أم لا، وبالتالي 5.0/4.0 هو 1.25، ولكن 5/4 هو 1 لأن قسمة العدد الصحيح تُقرب النتيجة الخاصة بالكسور للصفر.

لو كانت نتيجة العملية الحسابية، سواء كانت موقعة أو غير موقعة، بها بِتات أكثر مما يمكن تمثيله في نوع النتيجة، يُقال أنها في حالة فيض (overflow). إن البتات ذات المستوى الأعلى غير المتلائمة مع النتيجة تحذف بصمت. لو كان الرقم الأصلي هو نوع موقّع، فإن النتيجة قد تكون سالبة لو كان البِت في أقصى اليسار هو 1، كما هو الحال في مثال int8 هنا:

```
var u uint8 = 255
fmt.Println(u, u+1, u*u) // "255 0 1"
var i int8 = 127
fmt.Println(i, i+1, i*i) // "127 -128 1"
```

يمكن مقارنة عددين صحيحين من نفس النوع باستخدام معاملات المقارنة الثنائية أدناه، ونوع تعبير المقارنة هو قيمة منطقية (Boolean).

```
==    يساوي
!=    لا يساوي
<     أقل من
<=    أقل من أو يساوي
>     أكبر من
>=    أكبر من أو يساوي
```

في الواقع، كل القيم التي تنتمي للنوع الأساسي – قيم منطقية أو أرقام أو سلاسل نصية- قابلة للمقارنة، بمعنى أنه يمكن مقارنة قيمتين من نفس النوع باستخدام المعاملات == و != . علاوة على ذلك، فإن الأعداد الصحيحة، وأرقام الفاصلة العائمة، والسلاسل تُرتب بواسطة معاملات المقارنة. إن قيم الكثير من الأنواع الأخرى غير قابلة للمقارنة، ولا يتم ترتيب أنواع أخرى. سنقدم عند مقابلتنا لكل نوع القواعد التي تحكم قابلية قيمه للمقارنة.

يوجد أيضًا معاملات جمع وطرح أحادية:

```
+    unary positive (no    effect)
-    unary negation
```

بالنسبة للأعداد الصحيحة، تُعتبر x+ اختصار لـ 0+x ، و-x اختصار لـ 0-x ، وبالنسبة للأرقام المعقدة والنقطة العائمة فإن x+ هي x فقط، و x− هي معكوس x.

تُقدّم Go أيضا المعاملات الثنائية للبِت التالية، وأول أربعة منها تعامل معامِلاتها كأنماط بِت دون تصور لأي نقل حسابي أو علامة:

```
&    bitwise    AND
|    bitwise    OR
^    bitwise    XOR
&^   bit clear (AND NOT)
<<   left shift
>>   right shift
```

إن المعامل ^ هو OR (XOR) بِت حصري عند استخدامه كمعامل ثنائي، ولكن عند استخدامه كمعامل سابقة أحادي، يُصبح معكوس بِت أو مكمل بِت، بمعنى أنه يعيد قيمة كل بِت ولكن بمعامل معكوس. إن المعامل ^& هو بت حذف (AND NOT): ففي تعبير z = x &^ y كل بِت في z يصبح يساوي 0 إذا كان البِت y المناظر له هو 1، بخلاف ذلك، سيساوي بِت x المناظر له.

توضح الشفرة أدناه كيف يمكن استخدام عمليات بِت لتفسير قيمة uint8 كمجموعة مضغوطة وكفء من 8 بِتات مستقلة. وهي تستخدم فِعل Printf's %b لطباعة أعداد الرقم الثنائية، وتعدّل 08 الـ b% (ظرف الحال – adverb) لتقرب النتيجة ذات الأصفار بحيث يتكون 8 أعداد فقط.

```go
var x uint8 = 1<<1 | 1<<5
var y uint8 = 1<<1 | 1<<2
fmt.Printf("%08b\n", x) // "00100010", the set {1, 5}
fmt.Printf("%08b\n", y) // "00000110", the set {1, 2}
fmt.Printf("%08b\n", x&y) // "00000010", the intersection {1}
fmt.Printf("%08b\n", x|y) // "00100110", the union {1, 2, 5}
fmt.Printf("%08b\n", x^y) // "00100100", the symmetric difference {2, 5}
fmt.Printf("%08b\n", x&^y) // "00100000", the difference {5}
for i := uint(0); i < 8; i++ {
   if x&(1<<i) != 0 { // membership test
    fmt.Println(i) // "1", "5"
   }
}
fmt.Printf("%08b\n", x<<1) // "01000100", the set {2, 6}
fmt.Printf("%08b\n", x>>1) // "00010001", the set {0, 4}
```

(يوضح القسم 6.5 كيف يمكن لتطبيق مجموعات العدد الصحيح أن تكون أكبر بكثير من بايت واحد).

في عملية تغيير x<<n و x>>n ، يحدد المعامل n عدد مراكز بِت التي ستحول، ويجب أن تكون غير موقّعة، ويمكن أن يكون معامل x موقع أو غير موقع. من الناحية الحسابية، تغيير x<<n لليسار مكافئ للضرب في 2^n، بينما تغيير x>>n لليمين، مكافئ لدور القسمة على 2^n.

تملأ التغييرات اليسارية البِتات بالأصفار، وكذلك التغييرات اليمينية الخاصة بالأرقام غير الموقعة، ولكن التغييرات اليمينية للأرقام الموقعة تملأ البتات الفارغة بنسخ من بِت التوقيع/العلامة. ولهذا السبب، من المهم استخدام الحساب غير الموقع عندما تعامل العدد الصحيح كنمط بِت.

بالرغم من أن جو توفر أرقام وحسابات غير موقعة، إلا أننا نميل لاستخدام شكل int الموقع حتى للكميات التي لا يمكن أن تكون سالبة، مثل طول مصفوفة، بالرغم من أن uint قد تبدو اختيار مناسب أكثر. بالتأكيد، تعيد وظيفة len المدمجة int موقعة، كما هو الحال في هذه الحلقة التي تعلن عن ميداليات الجوائز بترتيب عكسي.

```
medals := []string{"gold", "silver", "bronze"}
for i := len(medals) - 1; i >= 0; i-- {
    fmt.Println(medals[i]) // "bronze", "silver", "gold"
}
```

سيكون البديل كارثيًا، فلو أعادت len رقم غير موقع، فإن i أيضًا ستكون uint، والشرط i >= 0 سيكون صحيحًا دائمًا حسب التعريف. بعد التكرار الثالث، الذي كان فيه i == 0، وعبارة --i ستجعل i لا تصبح 1-، بل تصبح أقصى قيمة لـ uint (كمثال 2^{64}-1)، كما أن تقييم الميداليات[i] سيفشل في زمن التشغيل، أو يهلع (panic) (انظر 5.9)، نتيجة محاولة الدخول لعنصر خارج حدود الشريحة.

لهذا السبب، تُستخدم الأرقام غير الموقعة في الغالب فقط عندما يكون معاملي بِت الخاصين بها أو المعاملات الحسابية المميزة الخاصين بها مطلوبين فقط، كما هو الحال عند تطبيق مجموعات بِت، وتحليل صيغ ملف ثنائي، أو في التلبيد (hashing) أو عِلم التعمية (cryptography). ولا تُستخدم عادة مع الكميات غير السالبة وحسب.

إن التحويل الصريح مطلوب بشكل عام لتحويل قيمة من نوع إلى آخر، ويجب أن تحتوي المعاملات الثنائية للحساب والمنطق (باستثناء الإزاحات) على معاملات من نفس النوع. بالرغم من أن هذا ينتج عنه من حين لآخر تعبيرات أطول، إلا أنه يزيل فئة كاملة من المشكلات كذلك، ويجعل البرامج سهلة الفهم أكثر.

كمثال من سياقات أخرى مألوفة، انظر هذا التسلسل:

```
var apples int32 = 1
var oranges int16 = 2
var compote int = apples + oranges // compile error
```

إن محاولة ترجمة هذه الإعلانات الثلاثة ينتج عنه رسالة خطأ:

```
invalid operation: apples + oranges (mismatched types int32 and int16)
```

إن عدم التوافق في النوع يمكن إصلاحه بطرق متعددة، أكثرها مباشرة هو تحويل أي شيء إلى نوع شائع:

```
var compote = int(apples) + int(oranges)
```

كما ذكرنا في القسم 2.5، لكل نوع T، تحول عملية التحويل T(x) القيمة x إلى النوع t لو كان التحويل مسموحا به. لا تتضمن العديد من عمليات تحويل عدد صحيح إلى عدد صحيح أي تغيير في القيمة، ويجب أن تخبر المترجم كيفية تفسير قيمة ما. ولكن التحويل الذي يضيق نطاق العدد الصحيح الكبير إلى عدد أصغر، أو التحويل من عدد صحيح إلى فاصلة عائمة أو العكس، يمكن أن يغير القيمة أو يقلل الدِقة:

```
f := 3.141 // a float64
i := int(f)
fmt.Println(f, i) // "3.141 3"
f = 1.99
fmt.Println(int(f)) // "1"
```

إن التحويل من فاصلة عائمة إلى عدد صحيح يحذف أي كسور، ويقرّب الكسر إلى الصفر. يجب أن تتجنب التحويلات التي يكون المعامل فيها خارج نطاق النوع المستهدف، لأن السلوك يعتمد على التطبيق.

```
f := 1e100     // a float64
i := int(f) // result is implementation-dependent
```

يمكن كتابة حروف العدد الصحيح ذات أي حجم أو أي نوع كأرقام عشرية عادية، أو كرقم ثماني لو كان يبدأ بـ 0، مثل 0666، أو كرقم سداسي لو كان يبدأ بـ 0x أو 0X، كما في 0xdeadbeef. يمكن كتابة الأرقام السداسية بحروف صغيرة أو كبيرة. ويبدو أن الأرقام الثمانية تستخدم هذه الأيام لهدف واحد فقط – تصاريح الملف في نظم POSIX – ولكن الأرقام العشرية السداسية تُستخدم على نطاق واسع للتأكيد على نمط بِت الخاص بالقيمة الرقمية للرقم.

عند طباعة الأرقام باستخدام حزمة fmt، يمكننا التحكم في العدد الجذري والتهيئة مع الأفعال %d و %o و %x، كما هو موضح في هذا المثال:

```
o := 0666
fmt.Printf("%d %[1]o %#[1]o\n", o) // "438 666 0666"
x := int64(0xdeadbeef)
fmt.Printf("%d %[1]x %#[1]x %#[1]X\n", x)
// Output:
// 3735928559 deadbeef 0xdeadbeef 0XDEADBEEF
```

لاحظ استخدام خدعتين من خدع fmt، عادة ما تحتوي سلسلة تهيئة Printf على أفعال % متعددة، والتي تحتاج لنفس عدد المعاملات الإضافي، ولكن "ظروف حال" [1] بعد % تخبر Printf باستخدام أول معامل مرارًا وتكرارًا. ثانيًا، ظرف # لـ o% أو x% أو X% يخبر Printf بحذف السابقة 0 أو 0x أو 0X بالترتيب.

تُكتب حروف Rune كحرف داخل اقتباس منفرد. وأبسط مثال على هذا هو حرف ASCII مثل 'a'، ولكن يمكن كتابة أي نقطة شفرة Unicode إما بشكل مباشر أو بخلوصات (escapes) رقمية، كما سنرى بعد فترة قصيرة:

تُطبع الـ Runes باستخدام c%، أو مع q% لو كان الاقتباس مطلوب:

```
ascii := 'a'
unicode := '国'
newline := '\n'
fmt.Printf("%d %[1]c %[1]q\n", ascii) // "97 a 'a'"
fmt.Printf("%d %[1]c %[1]q\n", unicode) // "22269 国 '国'"
fmt.Printf("%d %[1]q\n", newline)    // "10 '\n'"
```

3.2 أرقام الفاصلة العائمة - Floating-Point Numbers

تقدم لغة جو حجمان من أرقام الفاصلة العائمة هما float32 و float64. وتُعَد خصائصهم الحسابية محكومة بمعيار IEEE 754 الذي تطبقه كل المعالجات الحديثة.

تتراوح قيم هذه الأنواع الرقمية من قيم صغيرة جدًا إلى قيم ضخمة، ويمكن إيجاد حدود قيم الفاصلة العائمة في حزمة math. إن الثابت math.MaxFloat32، هو أكبر float32، وهو حوالي 3.4e38، و math.MaxFloat64 هو حوالي 1.8e308. أما أقل قيم موجبة فهي قريبة من 1.4e-45 و 4.9e-324 بالترتيب.

تُقدّم float32 حوالي ستة أرقام عشرية دقيقة تقريبًا، بينما تقدم float64 حوالي 15 رقم، ويجب تفضيل float64 في معظم الأغراض، لأن حسابات float32 تتراكم فيها الأخطاء بسرعة ما لم يكن الفرد شديد الحرص، وأصغر رقم صحيح موجب لا يمكن تمثيله بالضبط كـ float32 ليس كبيرًا:

```
var f float32 = 16777216 // 1 << 24
fmt.Println(f == f+1)      // "true"!
```

يمكن كتابة أرقام الفاصلة العائلة حرفيًا باستخدام الأرقام العشرية كالتالي:

```
const e = 2.71828 // (approximately)
```

يمكن حذف الأرقام قبل العلامة العشرية (0.707) أو بعدها (.1). من الأفضل كتابة الأرقام الصغيرة جدًا أو الكبيرة جدًا بتدوين علمي، مع كون الحرف e أو E يسبق الأس العشري.

```
const Avogadro = 6.02214129e23
const Planck = 6.62606957e-34
```

تُطبق قيم الفاصلة العائمة باستخدام فِعل %g في Printf، والذي يختار أكثر تمثيل مضغوط ذي دقة مناسبة، ولكن بالنسبة لجداول البيانات، قد تكون الأشكال %e (أس) أو %f (بدون أس) مناسبة أكثر. تسمح كل الأفعال الثلاثة بالتحكم في عرض الحقل والدقة الرقمية.

```
for x := 0; x < 8; x++ {
    fmt.Printf("x = %d eˣ = %8.3f\n", x, math.Exp(float64(x)))
}
```

تطبع الشفرة أعلاه قوى e مع ثلاثة أرقام عشرية دقيقة، مصطفة في حقل من 8 حروف:

```
x = 0          eˣ =   1.000
x = 1          eˣ =   2.718
x = 2          eˣ =   7.389
x = 3          eˣ =   20.086
x = 4          eˣ =   54.598
x = 5          eˣ =   148.413
x = 6          eˣ =   403.429
x = 7          eˣ =   1096.633
```

إضافة إلى الارتباط الكبير بين الوظائف الرياضية المعتاد، تقوم حزمة math بوظائف أخرى لإنشاء وكشف القيم المميزة التي يُعرّفها IEEE 754: المالانهايات الموجبة والسالبة، والتي تمثل أرقام ذات أحجام هائلة ناتجة عن القسمة على صفر، و NaN("ليست رقم")، وهي تمثل نتيجة العمليات الرياضية المخادِعة مثل 0/0 أو Sqrt(-1).

```
var z float64
fmt.Println(z, -z, 1/z, -1/z, z/z) //"0 -0 +Inf -Inf NaN"
```

تختبر الوظيفة math.IsNaN ما إذا كانت معطياتها قيمة غير رقمية، وتعيد math.NaN هذه القيمة. من المغري استخدام NaN كقيمة رقيبة (sentinel) في الحوسبة الرقمية، ولكن اختبار ما إذا كانت وظيفة حسابية معينة تساوي NaN هو أمر صعب لأن أي مقارنة مع NaN تكون سالبة false دائمًا:

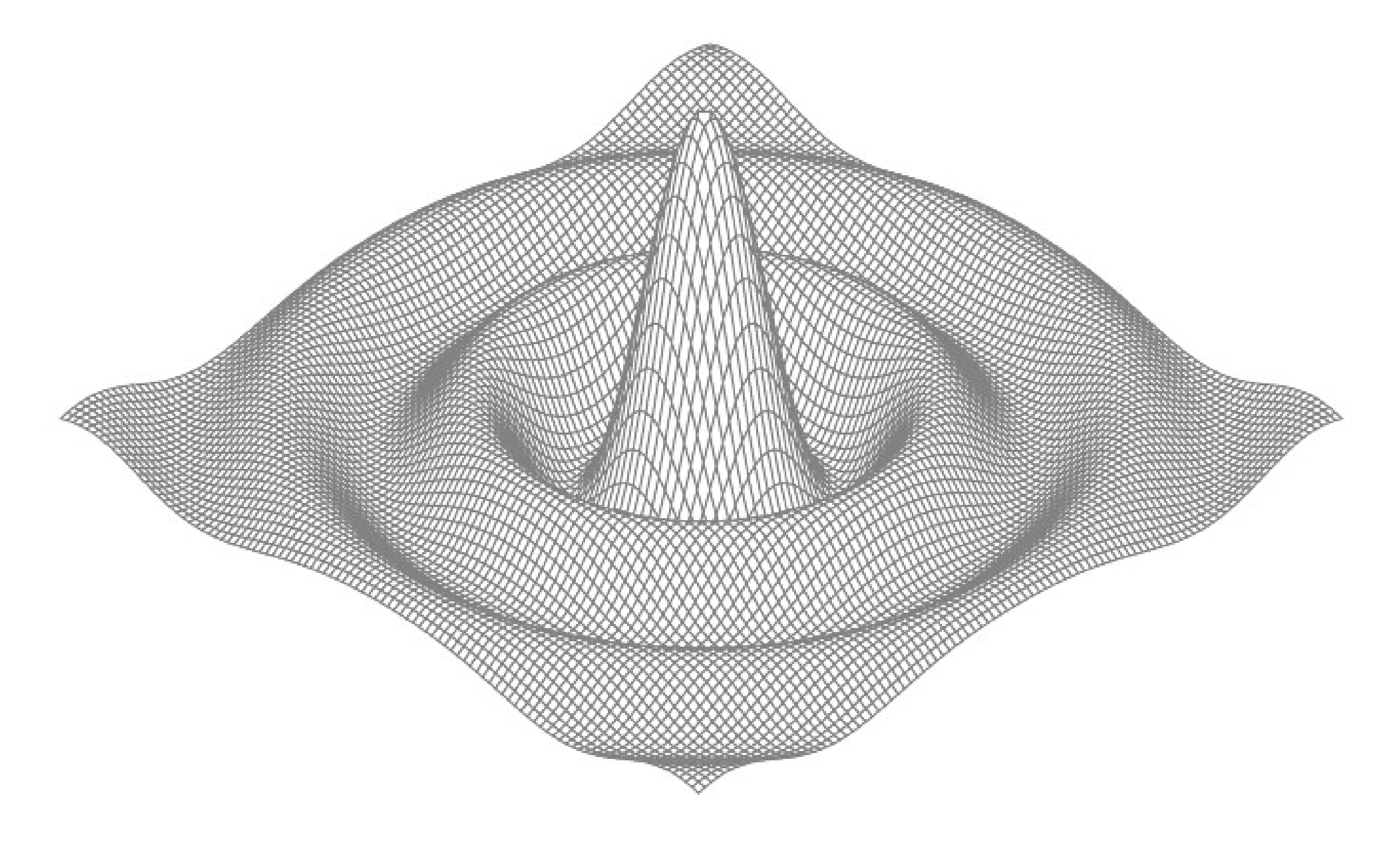
```
nan := math.NaN()
fmt.Println(nan == nan, nan < nan, nan > nan) // "false false false"
```

لو أن هناك وظيفة تعيد نتيجة فاصلة عائمة يمكن أن تفشل، فمن الأفضل تقديم تقرير الفشل بشكل منفصل، كالتالي:

```
func compute() (value float64, ok bool) {
```

```
// ...
     if failed {
          return 0, false
     }
     return result, true
}
```

يوضح البرنامج التالي حساب للفاصلة العائمة بالرسومات، وهو يخطط وظيفة من متغيرين z = f(x, y) كشبكة سلكية ذات سطح ثلاثي الأبعاد، باستخدام رسومات شعاعية متجهة (SVG)، وهي معيار XML قياسي للرسومات الخطية.



يوضح الشكل 3.1 مثال على ناتج الوظيفة sin(r)/r حيث r هي sqrt(x*x+y*y).

الشكل 3.1: مخطط سطحي للوظيفة sin(r)/r.

```
gopl.io/ch3/surface
// Surface computes an SVG rendering of a 3-D surface function.
package main
import (
   "fmt"
   "math"
)
```

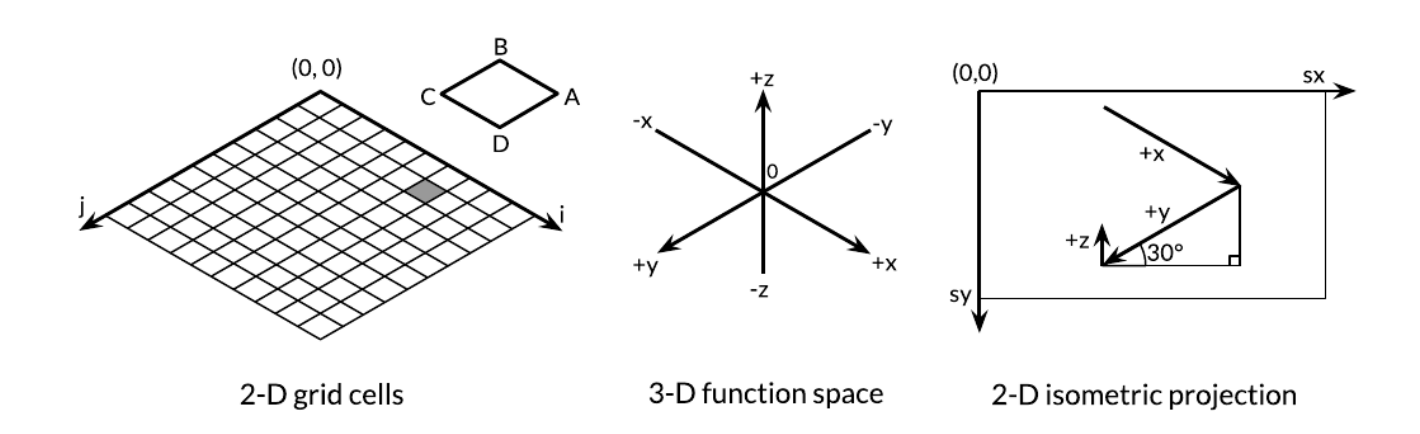
```
const (
    width, height = 600, 320            // canvas size in pixels
    cells         = 100                 // number of grid cells
    xyrange       = 30.0                // axis ranges (-xyrange..+xyrange)
    xyscale       = width / 2 / xyrange // pixels per x or y unit
    zscale        = height * 0.4        // pixels per z unit
    angle         = math.Pi / 6         // angle of x, y axes (=30°)
)
var sin30, cos30 = math.Sin(angle), math.Cos(angle) // sin(30°), cos(30°)
func main() {
    fmt.Printf("<svg xmlns='http://www.w3.org/2000/svg' "+
        "style='stroke: grey; fill: white; stroke-width: 0.7' "+
        "width='%d' height='%d'>", width, height)
    for i := 0; i < cells; i++ {
        for j := 0; j < cells; j++ {
            ax, ay := corner(i+1, j)
            bx, by := corner(i, j)
            cx, cy := corner(i, j+1)
            dx, dy := corner(i+1, j+1)
            fmt.Printf("<polygon points='%g,%g %g,%g %g,%g %g,%g'/>\n",
                ax, ay, bx, by, cx, cy, dx, dy)
        }
    }
    fmt.Println("</svg>")
}
func corner(i, j int) (float64, float64) {
    // Find point (x,y) at corner of cell (i,j).
    x := xyrange * (float64(i)/cells - 0.5)
    y := xyrange * (float64(j)/cells - 0.5)
    // Compute surface height z.
    z := f(x, y)
    // Project (x,y,z) isometrically onto 2-D SVG canvas (sx,sy).
    sx := width/2 + (x-y)*cos30*xyscale
    sy := height/2 + (x+y)*sin30*xyscale - z*zscale
    return sx, sy
}
func f(x, y float64) float64 {
    r := math.Hypot(x, y) // distance from (0,0)
    return math.Sin(r) / r
}
```

لاحظ أن الوظيفة corner تعيد قيمتين، وهما إحداثيات ركن الخلية.

يحتاج تفسير طريقة عمل البرنامج لمعرفة أساسيات الهندسة فقط، ولكن لا بأس بالمرور عليه بسرعة، حيث أن الهدف هو توضيح حساب الفاصلة العائمة. إن جوهر البرنامج هو الوصل بين ثلاث نظم إحداثية مختلفة، وهي الموضحة في الشكل 3.2. النظام الأول هو شبكة ثنائية الأبعاد (2-D) تتكون من 100×100 خلية تحددها إحداثيات الأعداد الصحيحة (i, j)، والتي تبدأ بـ (0 ,0) في الركن الخلفي البعيد. نحن نخطط من الخلف للأمام بحيث يمكن حجب مضلعات الخلفية بالمضلعات الأمامية.

إن النظام الإحداثي الثاني هو شبكة من إحداثيات فاصلة العائمة ثلاثية الأبعاد (3-D) (x, y, z)، حيث x و y هما دوال خطية لـ i و j، مترجمة بحيث يكون الأصل في المركز، وتتصاعد تدريجيًا بنطاق xy الثابت. إن الارتفاع z هو قيمة دالة السطح f (x, y).

أما النظام الإحداثي الثالث فهو رُقع الصورة ثنائية الأبعاد، حيث (0 ,0) في الركن العلوي الأيسر. إن النقاط في هذا المستوى يُكتب كـ (sx, sy). ونحن نستخدم الإسقاط متساوي الأبعاد لتخطيط النقطة ثلاثية الأبعاد (x, y, z) على الرقع ثنائية الأبعاد.



شكل (٣.٢) ثلاث نظم إحداثية مختلفة

تظهر نقطة في جزء أبعد من الجانب الأيمن للرقعة كلما كانت قيمة x الخاصة بها أكبر، أو قيمة y الخاصة بها أصغر. وتظهر النقطة على الجانب السفلي من الرقعة كلما كانت قيمة x أو y أكبر، وقيمة z أصغر. إن عوامل المقياس الرأسية والأفقية لـ x و y مشتقة من جيب وجيب تمام زاوية 30 درجة. وعامل المقياس z التي تساوي 0.4 هي قيمة عشوائية.

تحسب الوظيفة الأساسية الإحداثيات على رقع صورة الأركان الأربعة للمضلع ABCD، حيث B مناظرة للنقطة (i, j)، و A و C و D هم جيرانها، ثم تطبع تعليمات SVG لرسمها، وتفعل هذا في كل خلية في الشبكة ثنائية الأبعاد.

تمرين 3.1: لو أعادت الوظيفة f قيمة float64 لا نهائية، فإن ملف SVG سيحتوي على عناصر <polygon> غير صالحة (بالرغم من أن العديد من برامج عرض SVG تتعامل مع هذا ببراعة). عدّل البرنامج لتجاوز المضلعات غير الصالحة.

تمرين 3.2: جرّب على الصور المرئية الوظائف الأخرى في حزمة math. هل يمكنك إنتاج صندوق بيض، أو حيتان أو سِراج؟

تمرين 3.3: لوّن كل مضلع بناء على ارتفاعه، بحيث تكون أعاليه باللون الأحمر (ff0000#) ووديانه باللون الأزرق (#0000ff).

تمرين 3.4: بإتباع طريقة مثال Lissajous في القسم 1.7، قم ببناء خادم ويب يحسب الأسطح ويكتب بيانات SVG للعميل. يجب أن يحدد الخادم عنوان Content-Type هكذا:

w.Header().Set("Content-Type", "image/svg+xml")

(لم تكن هذه الخطوة مطلوبة في مثال Lissajous لأن الخادم يستخدم طرق كشف قياسية لمعرفة الهيئات الشائعة مثل PNG من أول 512 بايت في الاستجابة، وينتج عنوانا مناسبا). اسمح للعميل بتحديد قيم مثل الارتفاع والعرض واللون عند طلب HTTP للمعلمات.

3.3 الأرقام المركبة - Complex Numbers

تقدّم لغة جو حجمين من أحجام الأرقام المركبة وهما complex64 و complex128، ومكوناتهما هي float32 و float64 بالترتيب. تُنشئ وظيفة complex المُدمجة رقمًا من مكوناتها الحقيقية والتخيلية، ووظائف real و imag المُدمجة تستخلص تلك المُكونات:

```
var x complex128 = complex(1, 2) // 1+2i
var y complex128 = complex(3, 4) // 3+4i
fmt.Println(x*y)                 // "(-5+10i)"
fmt.Println(real(x*y))      // "-5"
fmt.Println(imag(x*y))      // "10"
```

لو كان حرف فاصلة عائمة أو حرف عدد صحيح عشري يليه i فورًا، مثل 3.141592i أو 2i، فإنه يصبح حرفا تخيليا (imaginary literal)، ويشير للرقم المركب ذي مُكوّن صفري حقيقي:

```
fmt.Println(1i * 1i) // "(-1+0i)", i² = -1
```

وفقًا لقواعد حساب الثابت (constant)، يمكن جمع ثوابت مركبة مع ثوابت أخرى (صحيحة أو فاصلة عائمة، تخيلية أو حقيقية)، مما يسمح بكتابة الأرقام المركبة بطريقة طبيعية، مثل 1+2i أو ما يكافئها، 2i+1. يمكن تبسيط إعلانات x و y أعلاه:

```
x := 1 + 2i
y := 3 + 4i
```

يمكن مقارنة الأرقام المركبة للمساواة من خلال == و =!. ويكون الرقمان المركبان متساويين إذا كانت أجزائهما الحقيقية متساوية، وأجزائهما التخيلية متساوية.

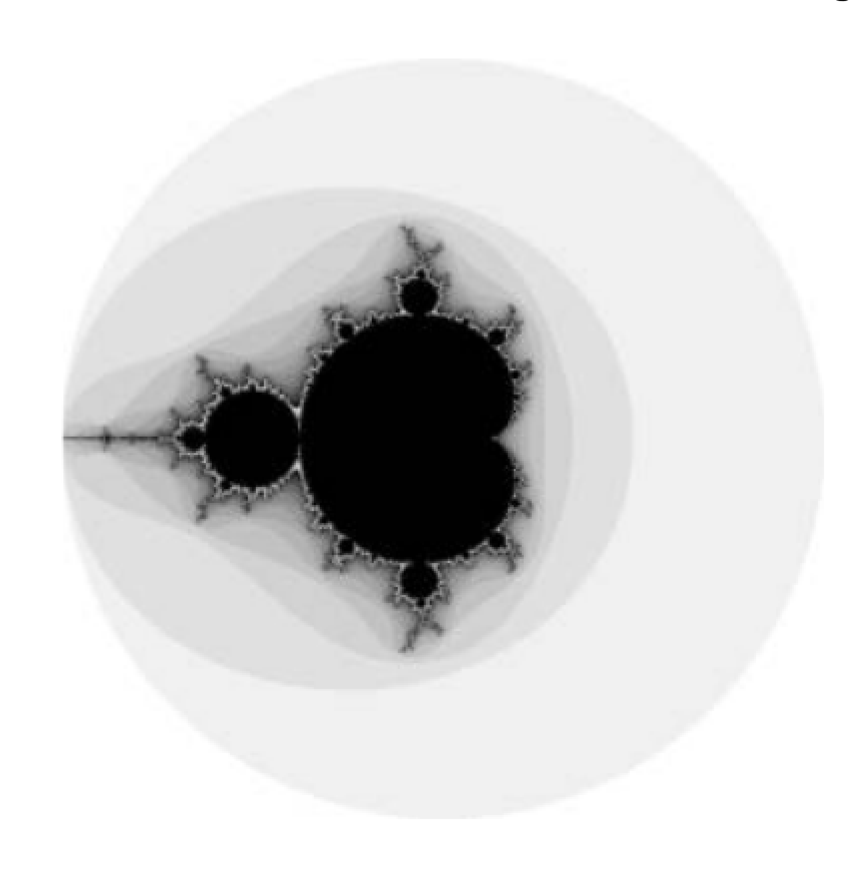
تقدم حزمة math/cmplx وظائف المكتبة للعمل على الأرقام المركبة، مثل الجذر التربيعي المركب، والوظائف الأسية.

```
fmt.Println(cmplx.Sqrt(-1)) // "(0+1i)"
```

يستخدم البرنامج التالي حساب complex128 لإنتاج مجموعة Mandelbrot:

```
gopl.io/ch3/mandelbrot
// Mandelbrot emits a PNG image of the Mandelbrot fractal.
package main
import (
    "image"
    "image/color"
    "image/png"
    "math/cmplx"
    "os"
)
func main() {
    const (
        xmin, ymin, xmax, ymax = -2, -2, +2, +2
        width, height          = 1024, 1024
    )
    img := image.NewRGBA(image.Rect(0, 0, width, height))
    for py := 0; py < height; py++ {
        y := float64(py)/height*(ymax-ymin) + ymin
        for px := 0; px < width; px++ {
            x := float64(px)/width*(xmax-xmin) + xmin
            z := complex(x, y)
            // Image point (px, py) represents complex value z.
            img.Set(px, py, mandelbrot(z))
        }
    }
    png.Encode(os.Stdout, img) // NOTE: ignoring errors
}
func mandelbrot(z complex128) color.Color {
    const iterations = 200
    const contrast = 15
    var v complex128
    for n := uint8(0); n < iterations; n++ {
        v = v*v + z
        if cmplx.Abs(v) > 2 {
            return color.Gray{255 - contrast*n}
        }
    }
    return color.Black
}
```

تتكرر الحلقتان المتداخلتان في كل نقطة في صورة نقطية بمقياس رمادي 1024x1024 تمثل الجزء -2 إلى +2 في المستوى المركب. يختبر البرنامج ما إذا كان تربيع وجمع الرقم الذي تمثله النقاط بشكل متكرر يؤدي في النهاية إلى "الهروب" من دائرة نصف قطرها 2. لو كان كذلك، فإن النقطة تظلل بعدد التكرارات التي تحتاجها للهروب. ولو لم يكن هذا هو الحال، فإن القيمة تنتمي إلى مجموعة Mandelbrot، وتظل النقطة سوداء. أخيرًا، يكتب البرنامج على مُخرَجه القياسي في الصورة المرمزة بـ PNG والخاصة بالكسر الأيقوني الموضح في الشكل 3.3.



الشكل 3.3: مجموعة Mandelbrot.

تمرين 3.5: طبّق مجموعة Mandelbrot كاملة اللون باستخدام الوظيفة image.NewRGBA والنوع color.RGBA أو color.YCbCr.

تمرين 3.6: إن Supersampling هي تقنية لتقليل تأثير البكسلة من خلال حساب قيمة اللون في نقاط متعددة داخل كل بيكسل وحساب المتوسط. أبسط طريقة هي قسمة كل بيكسل إلى أربع "بيكسلات فرعية- subpixels". طبّق هذه التقنية.

تمرين 3.7: يستخدم كسر بسيط آخر طريقة نيوتن لإيجاد حلول معقدة لوظيفة مثل $z^4-1 = 0$. ظلل كل نقطة بدء بواسطة عدد التكرارات المطلوبة للاقتراب من واحدة من الجذور الأربعة. لوّن كل نقطة حسب الجذر الذي تتعامل معه.

تمرين 3.8: إن تصيير (rendering) الكسور على مستويات تكبير مرتفعة يحتاج لدقة حسابية كبيرة. طبّق نفس الكسور باستخدام أربع تمثيلات مختلفة للأرقام هي: complex64 و complex128 و big.Float و big.Rat (يوجد آخر نوعين في حزمة math/big. تستخدم Float فاصلة دقة عائمة عشوائية ولكن مقيدة، بينما تستخدم Rat أرقام دقة منطقية غير مقيدة). كيف يمكن مقارنتهما من حيث الأداء واستخدام الذاكرة؟ ما هي مستويات التكبير التي تجعل أدوات التصيير مرئية؟

تمرين 3.9: اكتب خادم ويب يصيّر الكسور ويكتب بيانات الصورة للعميل. اسمح للعميل بتحديد x و y وكبّر القيم كمؤشرات لطلب HTTP.

3.4 القيم المنطقية - Booleans

إن القيمة من النوع bool أو boolean، لها قيمتين محتملتين فقط وهما true أو false. إن الشروط في عبارات if و for هما قيم منطقية، ومعاملات المقارنة مثل == و > ينتج عنها نتيجة منطقية. إن العامل الأحادي ! هو عَكس منطقي، وبالتالي !true هو false، أو يمكن القول أن true==(true==false!)، بالرغم من أننا لو اعتبرنا الأمر مسألة أسلوب، فسنبسّط دائمًا التعبيرات المنطقية الزائدة مثل x==true إلى x.

إن القيم المنطقية المُدمجة مع المشغلات && (AND) و || (OR)، التي تملك سلوك "قصير الدائرة" (short- circuit): لو كانت الإجابة محددة بالفعل بواسطة قيمة المعامل الأيسر، فإن المعامل الأيمن لا يُقيّم، مما يجعل من الآمن كتابة التعبيرات كالتالي:

```
s != "" && s[0] == 'x'
```

حيث s[0] ستصاب بالهلع لو طُبقت على سلسلة فارغة.

حيث أن && ذات نسبة أولوية أعلى من || (mnemonic: && هو مضاعفة للقيمة المنطقية، بينما || هي قيمة جمع للقيمة المنطقية)، غير مطلوب أي أقواس للشروط في هذا الشكل:

```
if 'a' <= c && c <= 'z' ||
    'A' <= c && c <= 'Z' ||
    '0' <= c && c <= '9' {
    // ...ASCII letter or digit...
}
```

لا يوجد تحويل ضمني من القيمة المنطقة إلى القيمة الرقمية مثل 0 أو 1 أو العكس، من الضروري استخدام if صريحة، كما هو الحال في:

```
i := 0
if b {
    i = 1
}
```

قد يستحق الأمر كتابة وظيفة تحويل لو كانت هذه العملية مطلوبة بشكل متكرر:

```
// btoi returns 1 if b is true and 0 if false.
```

```
func btoi(b bool) int {
    if b {
        return 1
    }
    return 0
}
```

إن العملية العكسية بسيطة جدًا لدرجة أنها لا تستحق وظيفة، ولكن لتقديم أمثلة متماثلة وحسب، ها هي ذا:

```
// itob reports whether i is non-zero.
func itob(i int) bool { return i != 0 }
```

3.5 السلاسل النصية - Strings

إن السلسلة النصية هي تسلسل بايتات غير قابل للتغيير، وقد تحتوي السلاسل على بيانات عشوائية، بما في ذلك بايتات قيمتها صفر، ولكنها عادة ما تتضمن نصا يمكن للبشر قراءته. إن سلاسل النص تُفسّر عادة كتسلسلات مُرمّزة بطريقة UTF-8 المنتمية لنقاط شفرة Unicode (runes)، والتي سنستكشفها بالتفصيل قريبًا.

تعيد وظيفة len المضمنة عدد البايتات (وليس ال runes) في السلسلة، بينما عملية s[i] الخاصة بالفهرس (index) تستعيد بايت i-th في السلسلة s، حيث 0 ≤ i < len(s).

```
s := "hello, world"
fmt.Println(len(s)) // "12"
fmt.Println(s[0], s[7]) // "104 119" ('h' and 'w')
```

إن محاولة الدخول للبايت خارج هذا النطاق ينتج عنه هلع:

```
c := s[len(s)] // panic: index out of range
```

إن البايت i-th الخاص بالسلسلة ليس بالضرورة حرف i-th في السلسلة، لأن ترميز UTF-8 للترميز غير الأسكي non-ASCII يتطلب اثنين من البايتات أو أكثر. سنناقش العمل على الحروف في الأجزاء التالية.

إن العملية s[i:j] الخاصة بالسلسلة الفرعية (substring) ينتج عنها سلسلة جديدة تتكون من بايتات السلسلة الأصلية التي تبدأ عند الفهرس i وتتواصل في الارتفاع حتى الفهرس j ولكنها لا تتضمنه. تحتوي النتيجة على البايتات j-i.

```
fmt.Println(s[0:5]) // "hello"
```

مرة أخرى، يحدث هلع لو كان المؤشر إما خارج الحدود أو كانت j أقل من i.

يمكن حذف أي من المعاملين i أو j أو كلاهما، وفي تلك الحالة يفترض أن القيم الافتراضية هي صفر (بداية السلسلة) و len(s) (نهايتها)، بالترتيب:

```
fmt.Println(s[:5])   // "hello"
fmt.Println(s[7:])   // "world"
fmt.Println(s[:])    // "hello, world"
```

إن المعامل + يصنع سلسلة جديدة من خلال الوصل بين سلسلتين.

```
fmt.Println("goodbye" + s[5:]) // "goodbye, world"
```

يمكن مقارنة السلاسل باستخدام معاملات المقارنة مثل == و <، وتتم المقارنة بايت ببايت، وبالتالي تكون النتيجة هي ترتيب معجمي طبيعي.

إن قيم السلسلة ثابتة وغير قابلة للتغيير: تسلسل البايت المتضمن في قيمة السلسلة لا يمكن أن يتغير أبدًا، بالرغم من أن بإمكاننا منح قيمة جديدة بالتأكيد لمتغير السلسلة (string variable). لو أردنا أن نصِل سلسلة بأخرى مثلًا، يمكن أن نكتب:

```
s := "left foot"
t := s
s += ", right foot"
```

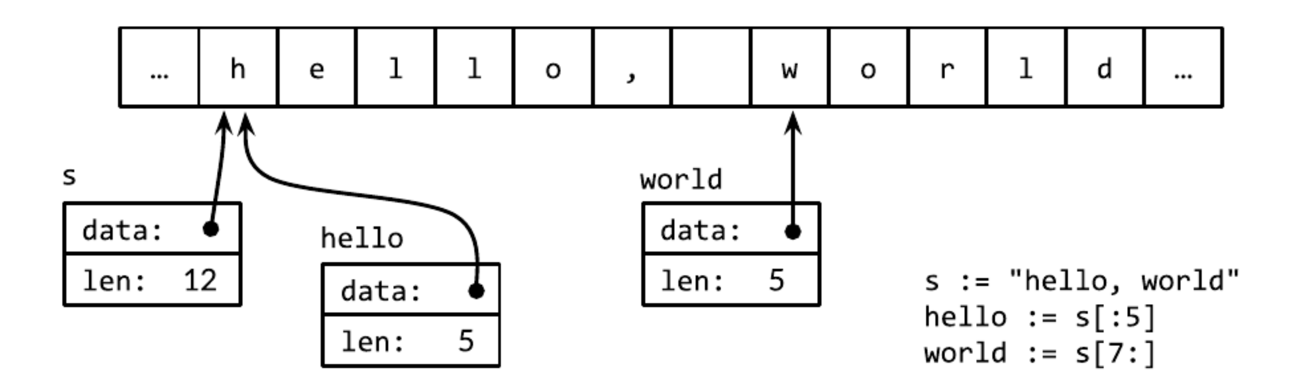
لا يعدّل هذا السلسلة التي تحملها s في الأصل، ولكنها تجعل s تحمل سلسلة جديدة تكونت من خلال عبارة =+، وفي تلك الأثناء، ستستمر t في احتواء السلسلة القديمة.

```
fmt.Println(s) // "left foot, right foot"
fmt.Println(t) // "left foot"
```

حيث أن السلاسل ثابتة، بالتالي غير مسموح بالتراكب التي تحاول تعديل بيانات السلسلة في مكانها:

```
s[0] = 'L' // compile error: cannot assign to s[0]
```

إن الثبات أو عدم القابلية للتغيير تعني أنه من الآمن أن تتشارك نسختان من السلسلة نفس الذاكرة الضمنية، مما يجعل نسخ السلاسل مهما كان طولها عملية رخيصة. بالمثل، السلسلة s وسلسلة فرعية مثل s[7:] يمكن أن تتشاركا نفس البيانات بأمان، وبالتالي فإن عملية السلسلة الفرعية أيضًا رخيصة. لا يوجد ذاكرة جديدة تُخصص في كلتا الحالتين. يوضح الشكل 3.4 ترتيبات السلسة واثنين من سلاسلها الفرعية التي تتشارك في نفس مصفوفة البايت الضمنية.



الشكل 3.4: السلسلة "hello, world" وسلسلتين فرعيتين.

3.5.1 حروف السلسلة – String Literals.

يمكن كتابة قيمة السلسلة كحرف سلسلة، وهو تسلسل من البايتات المتضمنة في اقتباسات مزدوجة:

```
"Hello, 世界"
```

نظرًا لكون ملفات مصدر Go مُرمزة دائمًا في UTF-8، وتُترجم سلاسل جو النصية عادة كـ UTF-8، يمكننا تضمين نقاط شفرة Unicode في حروف السلسلة.

يمكن استخدام "تسلسلات الخلوص" (escape sequences) التي تبدأ بـ \ داخل حروف السلسلة ذات علامات الاقتباس المزدوجة من أجل إدخال قيم بايت عشوائية إلى السلسلة. إن أحد مجموعات الخلوص تتعامل مع شفرات تحكم ASCII مثل سطر جديد ورجوع إلى السطر وعلامة تبويب:

```
\a  ‘‘alert’’ or bell
\b   backspace
\f   form feed
\n   newline
\r  carriage return
\t  tab
\v  vertical tab
\'  single quote (only in the rune literal '\'')
\"  double quote (only within "..." literals)
\\  backslash
```

يمكن إدراج البايتات الكيفية أيضًا في السلاسل الحرفية باستخدام خلوصات سداسية عشرية أو ثُمانية. إن الخلوص السداسي العشري (hexadecimal escape) يُكتب \xhh، مع رقميّ h سداسيين عشريين بالضبط (بحروف صغيرة أو كبيرة). ويُكتب الخلوص الثماني (octal escape) كـ \ooo بثلاث أرقام o ثمانية بالضبط (0 إلى 7) لا تتجاوز \377. يرمز كلاهما إلى بايت واحد بقيمة محددة. سنرى لاحقًا كيفية ترميز نقاط شفرة Unicode رقميًا في حروف السلسلة.

يُكتب حرف السلسلة الخام (raw string literal) كـ '...'، باستخدام علامات الاقتباس الخلفية بدلًا من الاقتباسات المزدوجة. لا يتم معالجة حرف تسلسلات خلوص في حرف السلسلة الخام، وتؤخذ المحتويات حرفيًا، بما في ذلك الخطوط المائلة العكسية (\) والسطور الجديدة، وبالتالي يمكن أن ينتشر حرف السلسلة الجديدة عبر سطور معينة في مصدر البرنامج. إن المعالجة الوحيدة هي أن عمليات الرجوع للسطر تُحذف بحيث تكون قيمة السلسلة واحدة في كل المنصات، بما في ذلك تلك التي تضع الرجوع للسطر في الملفات النصية بطريقة ملائمة.

إن حروف السلسلة الخام هي طريقة مريحة لكتابة تعبيرات منتظمة، والتي تحتوي عادة على الكثير من السطور المائلة الخلفية. كما أنها مفيدة كذلك في قوالب HTML، وحروف JSON، ورسائل استخدام الأمر، وما شابه ذلك، والتي تمتد عادة لأسطر متعددة.

```
const GoUsage = 'Go is a tool for managing Go source code.
Usage:
go command [arguments]
...
```

3.5.2 يونيكود – Unicode

منذ وقت طويل مضى، كانت الحياة بسيطة، ولو من منظور ضيق الأفق على الأقل، ولم يكن هناك سوى مجموعة حروف واحدة نتعامل معها: ASCII أو الشفرة القياسية الأمريكية لتبادل المعلومات (ASCII)، أو بدقة أكثر US-ASCII، والتي تستخدم 7 بِت لتمثيل 128 "حرف"، وهم: الحروف الكبيرة والصغيرة في اللغة الإنجليزية، والأرقام، ومجموعة متنوعة من حروف الترقيم والتحكم في الأدوات. كان هذا لا بأس به لفترة طويلة في الأيام المبكرة لابتكار الكمبيوتر، ولكنه ترك جزءًا كبيرًا جدًا من سكان العالم دون قدرة على استخدام نُظم كتابتهم الخاصة على أجهزة الكمبيوتر. ومع نمو الإنترنت، أصبحت البيانات المكتوبة بلغات كثيرة شائعة أكثر. كيف يمكن التعامل مع هذا التنوع الثري؟ وبكفاءة؟

إن الإجابة هي Unicode (unicode.org)، والتي تجمع كل الحروف في كل نظم الكتابة في العالم، إضافة إلى اللكنات، واللهجات الأخرى، وشفرات التحكم مثل علامة التبويب والرجوع للسطر، والكثير من الأشياء المقصورة على فئة معينة، ومنح كل منها رقمًا قياسيًا يُطلق عليه "نقطة شفرة يونيكود" (Unicode code point) أو Rune بلغة Go.

تُعرف النسخة 8 من Unicode نقاط الشفرة الخاصة بأكثر من 120000 حرف في أكثر من 100 لغة ونص. كيف تم تمثيلها في برامج الكمبيوتر والبيانات؟ إن نوع البيانات الطبيعي الذي يحمل rune واحد هو int32، وهذا هو ما تستخدمه Go، فهي تحتوي على rune المترادف لخدمة هذا الغرض بالضبط.

يمكننا تمثيل تسلسل من runes متسلسل من قيم int32. وفي هذا التمثيل، الذي يُطلق عليه UFT-32 أو UCS-4، يكون حجم ترميز كل نقطة شفرة Unicode واحدًا، وهو 32 بِت. إن هذا بسيط وموحد، ولكنه يستخدم مساحة أكبر بكثير مما

هو ضروري حيث أن معظم النص الذي يقرأه الكمبيوتر مكتوب بـ ASCII، والذي يحتاج فقط لـ 8 بِت أو 1 بايت لكل حرف. إن كل الحروف منتشرة الاستخدام عددها أقل من 65536، وهو ما يناسب الـ 16 بِت. هل يمكننا فعل ما هو أفضل من هذا؟

3.5.3 UTF-8

إن UTF-8 هو ترميز متغير الطول لنقاط شفرة Unicode كبايتات. وقد ابتكره كل من Ken Thompson و Rob Pike ، وهما أيضًا من واضعي لغة Go، وقد أصبح قياسيًا في Unicode الآن. يستخدم UTF-8 ما بين 1 و 14 بايت لتمثيل كل rune، ولكن 1 بايت فقط لكل حرف ASCII، و 2 أو 3 بايت فقط لمعظم ال runes شائعة الاستخدام. إن البِتات الأعلى مستوى للبايت الأولى في ترميز الـ rune تشير إلى عدد البايتات التالية. إن المستوى الأعلى 0 يشير إلى ASCII من 7 بِت، حيث يأخذ كل rune 1 بايت فقط، وبالتالي هو مطابق لـ ASCII التقليدي. يشير المستوى الأعلى 110 إلى أن ال rune يأخذ 2 بايت، ويبدأ البايت الثاني ب 10. وتحتوي ال runes الأكبر على ترميزات تناظرية.

```
0xxxxxx                             runes 0-127     (ASCII)
11xxxxx 10xxxxxx                    128-2047        (values <128 unused)
110xxxx 10xxxxxx 10xxxxxx           2048-65535      (values <2048 unused)
1110xxx 10xxxxxx 10xxxxxx 10xxxxxx  65536-0x10ffff  (other values unused)
```

إن الترميز متغير الطول يلغي الفهرسة المباشرة للدخول إلى حرف n-th في السلسلة، ولكن UTF-8 يحتوي على العديد من الخصائص المرغوب فيها التي تعوض عن ذلك. إن الترميز مضغوط، ومتوافق مع ASCII، ومتزامن ذاتيًا: من الممكن إيجاد بداية الحرف من خلال عمل نسخة احتياطية لما لا يزيد عن 3 بايت. وهو أيضًا شفرة سابقة (prefix)، وبالتالي يمكن فك ترميزها من اليسار إلى اليمين دون أي غموض أو تطلع للأمام. لا يُعتبر أي ترميز لـ rune سلسلة فرعية لأي ترميز آخر، أو حتى تسلسل للآخرين، وبالتالي يمكنك البحث عن rune من خلال البحث عن بايتاته وحسب، دون القلق بشأن السياق السابق له. إن الترتيب المعجمي للبايت مساوي لترتيب نقطة شفرة Unicode، وبالتالي فإن فرز UTF-8 يعمل بشكل طبيعي. لا يوجد بايتات NUL (صفر) مدمجة، وهو مناسب للغات البرمجة التي تستخدم NUL لإنهاء السلاسل.

تُرمز ملفات جو المصدرية دائمًا باستخدام UTF-8، و UTF-8 هو الترميز المفضل للسلاسل النصية التي تتلاعب بها برامج Go. تقدم حزمة unicode وظائف للعمل على runes الفردية (مثل التمييز بين الحروف والأرقام، أو تحويل الحروف الكبيرة إلى حروف صغيرة)، وتقدم حزمة unicode/utf8 وظائف لترميز وفك ترميز runes كبايتات باستخدام UTF-8.

إن العديد من حروف Unicode من الصعب كتابتها على لوحة المفاتيح أو تمييزها بصريًا عن حروف مشابهة لها، وبعضها غير مرئي حتى. إن خلوصات Unicode إلى حروف سلسلة go تسمح لنا بتحديدها حسب قيمة نقطة شفرتها الرقمية.

وهناك شكلين لها وهما: uhhhh\ للقيمة الـ 16 بِت، و Uhhhhhhhh\ للقيمة الـ 32 بِت، حيث كل h هي رقم سداسي عشري، وتنشأ الحاجة للشكل 32 بِت بطريقة متباينة وغير متكررة. يرمز كل منهم إلى ترميز UTF-8 لنقطة شفرة محددة. من ثم، كمثال، حروف السلسلة التالية تمثل كلها نفس السلسلة الـ 6 بايت.

```
"世界"
"\xe4\xb8\x96\xe7\x95\x8c"
"\u4e16\u754c"
"\U00004e16\U0000754c"
```

إن تسلسلات الخلوص الثلاثة أعلاه تقدم تدوينات بديلة لأول سلسلة، ولكن القيم التي ترمز لها متطابقة.

يمكن استخدام خلوصات Unicode أيضًا في حروف rune، وهذه الحروف الثلاثة مكافِئة لبعضها:

```
'世'      '\u4e16'   \U00004e16'
```

إن ال rune الذي قيمته أقل من 256 يمكن كتابته بخلوص سداسي عشري واحد، مثل 'x41\' لـ 'A'، ولكن يجب استخدام خلوص u\ أو U\ في القيم الأعلى. من ثم، لا يُعد 'xe4\xb8\x96\' حرف rune قانوني، بالرغم من أن الثلاثة بايت هذه هي ترميز UTF-8 مناسب لنقطة الشفرة المنفردة.

بفضل خصائص UTF-8 الجيدة حقًا، لا تحتاج العديد من عمليات السلسلة إلى فك الترميز. يمكننا اختبار ما إذا كانت سلسلة تحتوي على سلسلة أخرى كسابقة لها:

```
func HasPrefix(s, prefix string) bool {
    return len(s) >= len(prefix) && s[:len(prefix)] == prefix
}
```

أو كلاحقة لها:

```
func HasSuffix(s, suffix string) bool {
    return len(s) >= len(suffix) && s[len(s)-len(suffix):] == suffix
}
```

أو كسلسلة فرعية:

```
func Contains(s, substr string) bool {
    for i := 0; i < len(s); i++ {
        if HasPrefix(s[i:], substr) {
            return true
        }
    }
return false
```

```
}
```

باستخدام نفس منطق النص المُرمز بـ UTF-8 على البايتات الخام. هذا ليس صحيحًا بالنسبة للترميزات الأخرى. (الوظائف أعلاه مأخوذة من حزمة strings، بالرغم من أن تطبيقها لـ Contains يستخدم تقنية التبليد للبحث بكفاءة أكبر).

من ناحية أخرى، لو كنا نهتم حقًا بحروف Unicode الفردية، يجب أن نستخدم آليات أخرى. فكر في السلسلة الموجودة في مثالنا الأول، والتي تتضمن حرفين من شرق أسيا. يوضح الشكل 3.5 تمثيلهما في الذاكرة. تحتوي السلسلة على 13 بايت، ولكنها تُفسّر كـ UTF-8، وهي ترمز تسع نقاط فقط أو تسع runes:

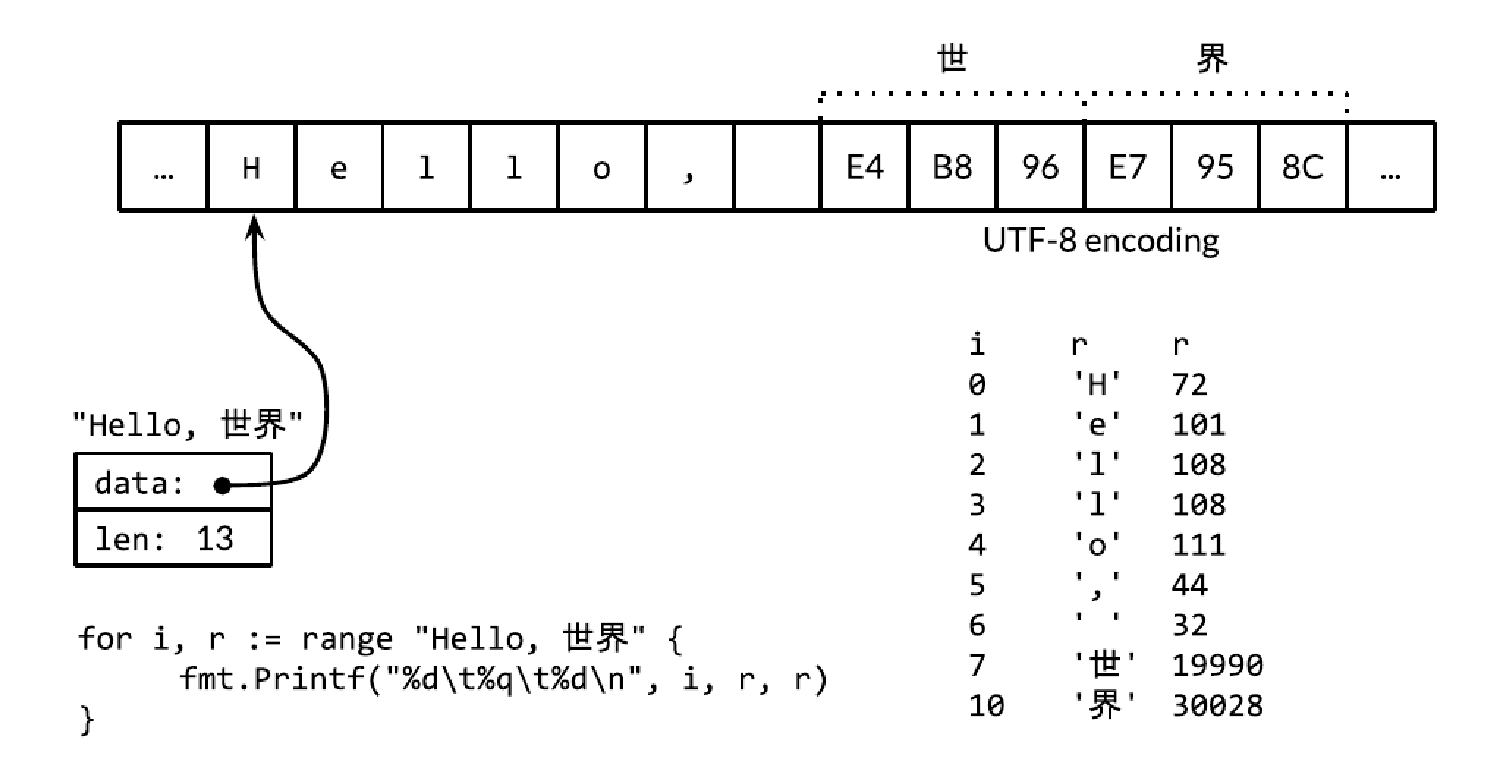
```
import "unicode/utf8"

s := "Hello, 世界"
fmt.Println(len(s))              // "13"
fmt.Println(utf8.RuneCountInString(s)) // "9"
```

لمعالجة هذه الحروف، سنحتاج لأداة فك ترميز UTF-8. تقدم حزمة unicode/utf8 أداة يمكننا استخدامها كالتالي:

```
for i := 0; i < len(s); {
    r, size := utf8.DecodeRuneInString(s[i:])
    fmt.Printf("%d\t%c\n", i, r) i += size
}
```

إن كل استدعاء لـ DecodeRuneInString يعيد r، ال rune نفسه، وحجم وعدد البايتات التي يشغلها ترميز UTF-8 الخاص بـ r. يُستخدم الحجم لتحديث مؤشر البايت i في ال rune التالي في السلسة. ولكن هذا فوضوي، ونحن بحاجة لحلقات من هذا النوع طوال الوقت. لحسن الحظ، تقوم حلقة نطاق Go بفك ترميز UTF-8 ضمنيًا عند تطبيقها على سلسلة. إن مُخرَج الحلقة أدناه موضح في الشكل 3.5 أيضًا، لاحظ كيف يقفز الفهرس بأكثر من 1 لكل rune لا ينتمي لـ ASCII (أو non-ASCII).



الشكل 3.5: حلقة نطاق تفك ترميز سلسلة مرمزة بـ UTF-8.

```
for i, r := range "Hello, 世界" {
    fmt.Printf("%d\t%q\t%d\n", i, r, r)
}
```

يمكننا استخدام حلقة نطاق بسيطة لحساب عدد الـ runes في السلسلة، كالتالي:

```
n := 0
for _, _ = range s {
    n++
}
```

كما هو الحال مع أشكال حلقة النطاق الأخرى، يمكننا حذف المتغيرات التي لا نحتاجها:

```
n := 0
for range s {
    n++
}
```

أو يمكننا استدعاء utf8.RuneCountInString(s) وحسب.

لقد ذكرنا سابقًا أن أغلب الأمر هو مسألة تقليد أو عرف في Go أن السلاسل النصية تُفسر كتسلسلات مرمزة بـ UTF-8 في نقاط شفرة Unicode، ولكن لاستخدام حلقات النطاق في السلاسل بشكل صحيح، سيكون الأمر ضروري وليس مجرد عُرف تقليدي. ماذا سيحدث لو مددنا نطاق على سلسلة تحتوي على بيانات ثنائية كيفية، أو بيانات UTF-8 بها أخطاء؟

في كل مرة تقوم فيها أداة فك ترميز UTF-8 باستهلاك بايت مُدخل غير مُدخل - سواء بشكل مباشر من خلال استدعاء utf8.DecodeRuneInString أو ضمني في حلقة نطاق، فإنها تقوم بإنتاج "حرف استبدال" Unicode مميز، '\uFFFD'، والذي يُطبع عادة كعلامة استفهام بيضاء داخل شكل سداسي أو ماسيّ الشكل�. عندما يواجه البرنامج قيمة rune هذه، فإنها عادة ما تكون علامة على أن أحد الأجزاء العلياء في البرنامج التي تنتج بيانات السلسلة النصية عولجت بإهمال في ترميزات النص.

إن UTF-8 مريحة بشكل استثنائي كتهيئة تبادلية، ولكن قد تكون ال runes مريحة أكثر داخل البرنامج لأنها ذات حجم موحدة، وبالتالي يمكن فهرستها في مصفوفات وشرائح بسهولة.

إن تحويل rune[] المُطبق على سلسلة مُرمزة بـ UTF-8 يعيد تسلسل نقاط شفرة Unicode التي ترمزها السلسلة.

```
// "program" in Japanese katakana
s := "プログラム"
fmt.Printf("% x\n", s) // "e3 83 97 e3 83 ad e3 82 b0 e3 83 a9 e3 83 a0"
r := []rune(s)
fmt.Printf("%x\n", r) // "[30d7 30ed 30b0 30e9 30e0]"
```

(إن الفعل x% في أول Printf يُدخل مساحة بين كل زوج من الأعداد السداسية).

لو تم تحويل شريحة runes إلى سلسلة، فإنها تنتج تسلسل من ترميزات UTF-8 لكل rune:

```
fmt.Println(string(r)) // "プログラム"
```

إن تحويل قيمة عدد صحيح إلى سلسلة يفسر العدد الصحيح كقيمة rune، وينتج عنه تمثيل UTF-8 لهذا الـ rune:

```
fmt.Println(string(65)) // "A", not "65"
fmt.Println(string(0x4eac)) // "京"
```

لو كان ال rune غير صالح، فإن الحرف البديل يحل محله:

```
fmt.Println(string(1234567)) // "�"
```

3.5.4 السلاسل وشرائح البايت ـ Strings and Byte Slices

إن هناك أربع حزم قياسية مهمة بشكل خاص للتلاعب بالسلاسل وهي: البايتات (bytes) والسلاسل (strings) و strconv و Unicode. تقدم حزمة strings هذه العديد من الوظائف التي تقوم ببحث واستبدال ومقارنة وتشذيب وفصل وضَم السلاسل.

تحتوي حزمة bytes على وظائف مشابهة للتلاعب بشرائح البايتات من النوع byte[]، والتي تشترك في بعض الخصائص مع السلاسل. ونظرًا لكون السلاسل ثابتة وغير قابلة للتغيير، فإن بناء سلاسل تدريجيًا يمكن أن يتضمن الكثير من التخصيص والنسخ. في مثل هذه الحالات، سيكون من الكفء أكثر استخدام النوع bytes.Buffer، الذي سنوضحه في الجزء التالي.

تقدم حزمة strconv وظائف لتحويل القيمة المنطقية وقيم العدد الصحيح والفاصلة العائمة مِن وإلى تمثيلات سلاسلهم النصية، ووظائف للسلاسل ذات أقواس الاقتباس والتي بدونها.

تقدم حزمة Unicode وظائف مثل IsDigit و IsLetter و IsUpper و IsLower لتصنيف ال runes. وتأخذ كل وظيفة معطى rune واحد، وتعيد قيمة منطقية. إن وظائف التحويل مثل ToUpper و ToLower تحوّل ال rune إلى الحالة المحددة لو كان حرفًا. تستخدم كل هذه الوظائف فئات Unicode القياسية الخاصة بالحروف والأرقام وغيرها. تحتوي حزمة strings على وظائف مشابهة، وتُسمى أيضًا ToUpper و ToLower، والتي تعيد سلسلة جديدة مع تطبيق التحويل المحدد على كل حرف في السلسلة الأصلي.

إن وظيفة basename أدناه مستلهمة من أداة صدفة يونكس (Unix shell utility) Unix التي تحمل نفس الاسم. وفي نسختنا، تمسح basename(s) أي سابقة لـ s تبدو كمسار الملف ذي مكونات يفصل بينها علامات مائلة، وتزيل أي لاحقة تبدو كنوع ملف:

```
fmt.Println(basename("a/b/c.go")) // "c"
fmt.Println(basename("c.d.go"))      // "c.d"
fmt.Println(basename("abc"))    //    "abc"
```

تقوم النسخة الأولى من basename بكل العمل دون مساعدة المكتبات:

```
gopl.io/ch3/basename1
// basename removes directory components and a .suffix.
// e.g., a => a, a.go => a, a/b/c.go => c, a/b.c.go => b.c
func basename(s string) string {
   // Discard last '/' and everything before.
   for i := len(s) - 1; i >= 0; i-- {
      if s[i] == '/' {
         s = s[i+1:]
         break
      }
   }
   // Preserve everything before last '.'.
   for i := len(s) - 1; i >= 0; i-- {
      if s[i] == '.' {
         s = s[:i]
         break
      }
   }
   return s
```

```
}
```

تستخدم النسخة الأبسط وظيفة مكتبة strings.LastIndex:

```
func basename(s string) string {
    slash := strings.LastIndex(s, "/") // -1 if "/" not found
    s = s[slash+1:]
    if dot := strings.LastIndex(s, "."); dot >= 0 {
        s = s[:dot]
    }
    return s
}
```

تقدم حزم path و path/filepath مجموعة عامة أكثر من الوظائف الخاصة بالتلاعب بالأسماء الهرمية. وتعمل حزمة path مع المسارات المحددة بخط مائل على أي منصة. ولا يجب استخدامها في أسماء الملفات، ولكنها مناسبة للنطاقات الأخرى، مثل مُكوّن المسار في URL. على النقيض، تتلاعب حزمة path/filepath بأسماء الملفات باستخدام قواعد لمنصة المستضيف، مثل /foo/bar لـ POSIX أو c: \foo\bar في ميكروسوفت ويندوز.

لنرى مثال سلسلة فرعية آخر. إن المهمة هي أخذ تمثيل سلسلة للعدد الصحيح، مثل "12345"، وإدخال الفاصلات (commas) بين كل ثلاث أماكن، كما في "12,345". تعمل هذه النسخة فقط على الأعداد الصحيحة، أما التعامل مع أرقام الفاصلة العائمة فسنقدمه كتمرين.

```
gopl.io/ch3/comma
// comma inserts commas in a non-negative decimal integer string.
func comma(s string) string {
    n := len(s)
    if n <= 3 {
        return s
    }
    return comma(s[:n-3]) + "," + s[n-3:]
}
```

إن المعطى لـ comma هو سلسلة. ولو كان الطول أقل من أو يساوي 3، لن تكون الفاصلة ضرورية. بخلاف ذلك، تستدعي الفاصلة نفسها بشكل متكرر عبر سلسلة فرعية تتكوّن من كل الحروف ما عدا آخر ثلاثة، وتصِل الفاصلة وآخر ثلاثة حروف بنتيجة الاستدعاء المتكرر.

تحتوي السلسلة على مصفوفة بايتات تصبح ثابتة وغير قابلة للتغيير بمجرد إنشائها. وعلى النقيض، يمكن تعديل عناصر شريحة بايتات بحرية.

يمكن تحويل السلاسل إلى شرائح بايت والعكس:

```
s := "abc"
b := []byte(s)
```

```
s2 := string(b)
```

بالتبعية، يخصص تحويل byte(s)[] مصفوفة بايت جديدة تحمل نسخة من بايتات s، وينتج عنها شريحة تشير إلى المصفوفة بأكملها. إن تحسين المترجم يمكن أن يساعد على تجنب التخصيص والنسخ في بعض الحالات، ولكن بصفة عامة، سيكون النسخ مطلوبا لضمان أن بايتات s ستظل دون تغيير حتى لو عُدلت بايتات b نتيجة هذا. إن التحويل من شريحة بايت إلى سلسلة باستخدام string(b) يؤدي لصنع نسخة أيضًا، لضمان ثبات سلسلة s2 الناتجة وعدم قابليتها للتغيير.

لتجنب التحويل وتخصيص الذاكرة غير الضروري، توازي العديد من وظائف المنفعة في حزمة bytes نظيراتها في حزمة string مباشرة. كمثال، إليك نصف دستة وظائف من strings:

```
func Contains(s, substr string) bool
func Count(s, sep string) int
func Fields(s string) []string
func HasPrefix(s, prefix string) bool
func Index(s, sep string) int
func Join(a []string, sep string) string
```

والمناظِرة لها من bytes:

```
func Contains(b, subslice []byte) bool
func Count(s, sep []byte) int
func Fields(s []byte) [][]byte
func HasPrefix(s, prefix []byte) bool
func Index(s, sep []byte) int
func Join(s [][]byte, sep []byte) []byte
```

الفارق الوحيد هو أن strings استُبدلت بشرائح البايت.

تقدم حزمة bytes نوع Buffer للتلاعب الكفء بشرائح البايت. ويبدأ Buffer فارغًا ولكنه ينمو مع كتابة بيانات عليه من أنواع مثل string و byte و []byes. كما يوضح المثال أدناه، لا يحتاج متغير bytes.Buffer إلى تمهيد لأن قيمته الصفرية قابلة للاستخدام:

```
gopl.io/ch3/printints
// intsToString is like fmt.Sprint(values) but adds commas.
func intsToString(values []int) string {
    var buf bytes.Buffer
    buf.WriteByte('[')
    for i, v := range values {
        if i > 0 {
            buf.WriteString(", ")
        }
        fmt.Fprintf(&buf, "%d", v)
```

```
    }
    buf.WriteByte(']')
    return buf.String()
}
func main() {
    fmt.Println(intsToString([]int{1, 2, 3})) // "[1, 2, 3]"
}
```

عند وصل ترميز UTF-8 الخاص بـ rune كيفي مع bytes.Buffer، من الأفضل استخدام طريقة WriteRune الخاصة بـ bytes.Buffer، ولكن WriteRune لا بأس بها لحروف ASCII مثل '[' و ']'.

إن نوع bytes.Buffer متنوع بشدة، وعندما نناقش الواجهات في الفصل السابع، سنرى كيف يمكن استخدامه كبديل للملف كلما كانت وظيفة I/O بحاجة إلى بالوعة (sink) للبايت (io.Writer) كما تفعل Fprintf أعلاه، أو كمصدر للبايتات (io.Reader).

تمرين 3.10: اكتب نسخة من برنامج comma غير تكرارية، باستخدام bytes.Buffer بدلًا من وصل السلاسل.

تمرين 3.11: حسّن comma بحيث تتعامل بشكل صحيح مع الأرقام الفاصلة العائمة ومع علامة اختيارية.

تمرين 3.12: اكتب وظيفة تقدم تقرير حول ما إذا كانت سلسلتين يعتبران جِناس (anagrams) لبعضهما، أي أن كلاهما يحتوي على نفس الحروف ولكن بترتيب مختلف.

3.5.5 التحويل بين السلاسل والأرقام ـ Conversions between Strings and Numbers

إضافة إلى التحويل بين السلاسل وال runes والبايتات، عادة ما يكون من الضروري التحويل بين القيم الرقمية وبين ما يماثلها في السلاسل النصية. يتم هذا بوظائف من حزمة strconv.

إن استخدام fmt.Sprintf هو أحد الخيارات المتاحة لتحويل عدد صحيح إلى سلسلة،، وهناك خيار آخر هو استخدام الوظيفة strconv.Itoa (لتحويل عدد صحيح إلى ASCII).

```
x := 123
y := fmt.Sprintf("%d", x)
fmt.Println(y, strconv.Itoa(x)) // "123 123"
```

يمكن استخدام FormatInt و FormatUint لتهيئة الأرقام في قاعدة مختلفة:

```
fmt.Println(strconv.FormatInt(int64(x), 2)) // "1111011"
```

إن أفعال fmt.Printf مثل b% و d% و u% و x% عادة ما تكون ملائمة أكثر من وظائف Format، وخاصة لو كنا نرغب في إدراج معلومات إضافية بجانب الرقم:

```
s := fmt.Sprintf("x=%b", x) // "x=1111011"
```

لتحليل سلسلة تمثل عدد صحيح، استخدام وظائف strconv مثل Atoi أو ParseInt أو ParseUint مع الأعداد الصحيحة غير المُوقّعة:

```
x, err := strconv.Atoi("123")             // x is an int
y, err := strconv.ParseInt("123", 10, 64)  // base 10, up to 64 bits
```

إن المعطى الثالث في ParseInt يمنح حجم نوع العدد الصحيح الذي يجب أن تناسبه النتيجة، كمثال 16 تعني ضمنًا int16، والقيمة المميزة لـ 0 تعني ضمنًا int. وعلى أي حال، نوع النتيجة y هو int64 دائمًا، والتي يمكنك تحويلها بعد ذلك إلى نوع أصغر.

تكون fmt.Scanf مفيدة أحيانًا لتحليل المُدخل الذي يتكون من أمزجة سلاسل وأرقام منظمة كلها في سطر واحد، ولكنها يمكن أن تكون غير مرنة، خاصة عند التعامل مع مُدخل غير كامل أو غير منتظم.

3.6 الثوابت - Constants

إن الثوابت هي تعبيرات قيمتها معروفة للمترجم، وتقييمها مضمون حدوثه في وقت التجميع وليس زمن التشغيل. إن النوع الضمني لكل ثابت هو النوع الأساسي: قيمة منطقية أو سلسلة أو رقم.

يُعرف إعلان const القيم المسماة التي تبدو شبيهة منهجيًا بالمتغيرات ولكن قيمتها ثابتة، والتي تمنع التغيير العرضي (أو الشائن) أثناء تنفيذ البرنامج. كمثال، الثابت مناسب أكثر من المتغير لثابت رياضي مثل pi، حيث أن قيمته لن تتغير:

```
const pi = 3.14159 // approximately; math.Pi is a better approximation
```

كما هو الحال مع المتغيرات، فإن تسلسل الثوابت يمكن أن يظهر في إعلان واحد، وسيكون هذا مناسب لمجموعة من القيم المرتبطة:

```
const (
e= 2.71828182845904523536028747135266249775724709369995957496696763
pi= 3.14159265358979323846264338327950288419716939937510582097494459
)
```

يمكن تقييم الكثير من الحسابات الخاصة بالثوابت بالكامل في وقت الترجمة، مما يقلل من العمل الضروري في زمن التشغيل، ويسمح بإجراء تحسينات أخرى للمترجم. يمكن ذِكر الأخطاء التي تُكشف عادة في زمن التشغيل في وقت الترجمة، عندما تكون معاملاتها ثوابت، مثل قسمة العدد الصحيح على الصفر، وفهرسة سلسلة خارج الحدود، وأي عملية فاصلة عائمة سينتج عنها قيمة لا نهائية.

إن نتائج كل العمليات الحسابية والمنطقية وعمليات المقارنة المُطبقة على معاملات الثابت هي في حد ذاتها ثوابت، وكذلك نتائج التحويلات واستدعاءات وظائف مدمجة معينة مثل len و cap و real و imag و complex و unsafe.Sizeof (انظر 13.1).

نظرًا لكون قيمها معروفة للمترجم، يمكن أن تظهر التعابير الثابتة في الأنواع، وخاصة كطول نوع مصفوفة:

```
const IPv4Len = 4
// parseIPv4 parses an IPv4 address (d.d.d.d).
func parseIPv4(s string) IP {
     var p [IPv4Len]byte
     // ...
}
```

إن إعلان الثابت يمكن أن يحدد النوع وكذلك القيمة، ولكن في غياب النوع الصريح، يُستدل على النوع من التعبير على الجانب الأيمن. وفي التالي، يُعتبر time.Duration نوع مسمى، ونوعه الضمني هو int64، و time.Minute هو ثابت من هذا النوع. من ثم، فإن كل ثابت من الثابتين المُعلنين أدناه كلاهما من النوع time.Duration أيضًا، كما تكشف T%:

```
const noDelay time.Duration = 0
const timeout = 5 * time.Minute
fmt.Printf("%T %[1]v\n",noDelay)          // "time.Duration 0"
fmt.Printf("%T %[1]v\n",timeout)          // "time.Duration 5m0s
fmt.Printf("%T %[1]v\n",time.Minute) // "time.Duration 1m0s"
```

عند إعلان تسلسل من الثوابت كمجموعة، فإن التعبير على الجانب الأيمن يمكن حذفه في كل شيء ما عدا أول المجموعة، مما يعني ضمنًا أن التعبير السابق ونوعه يجب استخدامه مرة أخرى. كمثال:

```
const (
    a = 1
    b
    c = 2
    d
)
fmt.Println(a, b, c, d) // "1 1 2 2"
```

إن هذا غير مفيد بشكل كبير لو كان تعبير الجانب الأيمن الذي يُنسخ ضمنًا يُقيّم دائمًا بنفس الشيء. ولكن ماذا لو كان يمكن أن يتباين؟ سيجلبنا هذا إلى iota.

3.6.1 iota المُنتِجة للثابت – The Constant Generator iota

يمكن لإعلان الثابت استخدام iota "المنتِجة للثابت"، والتي تُستخدم لإنشاء تسلسل من القيم المرتبطة دون التنصيص على كل واحدة منها بشكل صريح. وفي إعلان const، تبدأ قيمة iota عند الصفر، وتزيد بواحد لكل عنصر في التسلسل.

إليك مثال من حزمة time، والذي يُعرّف الثوابت المسماة للنوع Weedays لأيام الأسبوع، ويبدأ بصفر ليوم الأحد. إن الأنواع من هذا النوع يُطلق عليها عادة enumerations أو enums للاختصار:

```
type Weekday int
const (
   Sunday Weekday = iota
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
   Saturday
)
```

يعلن هذا أن يوم الأحد هو صفر والاثنين 1، إلخ.

يمكننا استخدام iota في تعبيرات أكثر تعقيدًا أيضًا، كما هو الحال في المثال من حزمة net، حيث كل بِت من أدنى 5 بِتات في العدد الصحيح غير الموقع يحصل على اسم مميز وتفسير للقيمة المنطقية:

```
type Flags uint
const (
     FlagUp Flags = 1 << iota // is up
     FlagBroadcast            // supports broadcast access capability
     FlagLoopback             // is a loopback interface
     FlagPointToPoint         // belongs to a point to point link
     FlagMulticast            // supports multicast access capability
)
```

كزيادات تدريجية في iota، فإن كل ثابت يُمنح قيمة 1 << iota، والتي تُقيّم لقوى متتابعة من اثنين، كل منها متوافق مع بِت واحد. يمكننا استخدام هذه الثوابت داخل وظائف تختبر أو تضبط أو تُخلي بِت أو أكثر من هؤلاء:

```
gopl.io/ch3/netflag
func IsUp(v Flags) bool     { return v&FlagUp == FlagUp }
func TurnDown(v *Flags)     { *v &^= FlagUp }
func SetBroadcast(v *Flags) { *v |= FlagBroadcast }
func IsCast(v Flags) bool   { return v&(FlagBroadcast|FlagMulticast) != 0 }
func main() {
   var v Flags = FlagMulticast | FlagUp
   fmt.Printf("%b %t\n", v, IsUp(v)) // "10001 true"
   TurnDown(&v)
```

```
    fmt.Printf("%b %t\n", v, IsUp(v)) // "10000 false"
    SetBroadcast(&v)
    fmt.Printf("%b %t\n", v, IsUp(v))   // "10010 false"
    fmt.Printf("%b %t\n", v, IsCast(v)) // "10010 true"
}
```

كمثال معقد أكثر على iota، يسمى هذا الإعلان قوى 1024:

```
const (
    _ = 1 << (10 * iota)
    KiB  // 1024
    MiB  // 1048576
    GiB  // 1073741824
    TiB  // 1099511627776              (exceeds   1 << 32)
    PiB  // 1125899906842624
    EiB  // 1152921504606846976
    ZiB  // 1180591620717411303424     (exceeds   1 << 64)
    YiB  // 1208925819614629174706176
)
```

إن آلية iota لها حدودها. كمثال، لا يمكن إنتاج قوى أكبر من الـ 1000 المألوفة (كيلوبايت، ميجا بايت، إلخ) نتيجة عدم وجود عامل أسي.

تمرين 3.13: اكتب إعلانات const للكيلو بايت (KB) والميجا بايت (MB) حتى YB بأكثر شكل مضغوط ممكن.

3.6.2 الثوابت التي بدون نوع – Untyped Constants

إن الثوابت في Go غير معتادة إلى حد ما، وبالرغم من أن الثابت يمكن أن يحتوي على أي نوع من أنواع البيانات الأساسية مثل int أو float64، بما في ذلك الأنواع الأساسية المُسماة مثل time.Duration، إلا أن العديد من الثوابت غير ملتزمة بنوع محدد. يمثل المترجم هذه الثوابت غير الملتزمة بدقة رقمية أكبر بكثير من قيم الأنواع الأساسية، والحساب الموجود فيهم أدق بكثير من الحساب الآلي. يمكنك افتراض وجود مستوى دقة 256 بِت على الأقل. هناك ست نكهات للثوابت غير الملتزمة هذه، ويُطلق عليها القيمة المنطقية بدون نوع (untyped boolean)، والعددي الصحيح بدون نوع (untyped integer)، والـ rune بدون نوع (untyped rune)، والفاصلة العائمة بدون نوع (untyped floating-point)، والمُركّب بدون نوع (untyped complex)، والسلسلة بدون نوع (untyped string).

ومن خلال تأجيل هذا الالتزام، تحتفظ الثوابت التي بدون نوع بدقتها الأعلى لوقت لاحق، وليس هذا وحسب، بل إن بإمكانها المشاركة في تعبيرات أكثر بكثير من الثوابت الملتزمة دون الحاجة لتحويلات. كمثال، القيم ZiB و YiB في المثال أعلاه أكبر من أن تُخزن في أي متغير عدد صحيح، ولكنها ثوابت سليمة يمكن استخدامها في تعبيرات كالتعبير التالي:

```
fmt.Println(YiB/ZiB) // "1024"
```

كمثال آخر، ثابت الفاصلة العائمة math.Pi يمكن استخدامه عند الحاجة لأي فاصلة عائمة أو قيمة مركبة:

```
var x float32 = math.Pi
var y float64 = math.Pi v
ar z complex128 = math.Pi
```

لو كان math.Pi ملتزم بنوع معين مثل float64، فإن النتيجة لن تكون بمثل هذه الدقة، وستحتاج تحويلات النوع لاستخدامه عندما نرغب في قيمة float32 أو complex128:

```
const Pi64 float64 = math.Pi
var x float32 = float32(Pi64)
var y float64 = Pi64
var z complex128 = complex128(Pi64)
```

أما بالنسبة للحروف، فإن الصياغة تحدد النكهة. فالحروف 0 و 0.0 و 0i و '\u0000' ترمز كلها لثوابت ذات نفس القيمة ولكن بنكهات مختلفة: عدد صحيح بدون نوع، وفاصلة عائمة بدون نوع، ومركب بدون نوع، و rune بدون نوع، بالترتيب. بالمثل، true و false هما قيم منطقية دون نوع، وحروف السلسلة هي سلاسل بدون نوع.

تذكّر أن / يمكن أن تمثل عدد صحيح أو قسمة فاصلة عائمة، وأن هذا يعتمد على معاملاتها. بالتبعية، يمكن لاختيار الحرف أن يؤثر على نتيجة تعبير قسمة الثابت:

```
var f float64 = 212
fmt.Println((f - 32) *5 / 9)        // "100"; (f - 32) * 5 is a float64
fmt.Println(5 / 9 * (f - 32))      // "0"; 5/9 is an untyped integer, 0
fmt.Println(5.0 / 9.0 * (f - 32))  // "100"; 5.0/9.0 is an untyped float
```

إن الثابت فقط هو الذي يمكن أن يكون دون نوع. وعندما يُعين ثابت بدون نوع إلى متغير، كما هو الحال في العبارة الأولى أدناه، أو يظهر على الجانب الأيمن لإعلان متغير ذو نوع صريح، كما هو الحال في العبارات الثلاثة الأخرى، فإن الثابت يُحول ضمنًا إلى نوع المتغير لو أمكن.

```
var f float64 = 3+0i   // untyped complex -> float64
f = 2                       // untyped integer  -> float64
f = 1e123             // untyped floating-point -> float64
f = 'a'               // untyped rune -> float64
```

من ثمّ، فإن العبارات أعلاه مكافئة للعبارات التالية:

```
var f float64 = float64(3 + 0i)
f = float64(2)
f = float64(1e123)
f = float64('a')
```

إن تحويل الثابت من نوع إلى آخر سواء بشكل صريح أو ضمني يحتاج أن يتمكن النوع الهدف من تمثيل القيمة الأصلية. إن التقريب مسموح به في أرقام الفاصلة العائمة الحقيقية والمركبة:

```
const (
    deadbeef = 0xdeadbeef  // untyped int with value 3735928559
    a = uint32(deadbeef)   // uint32 with value 3735928559
    b = float32(deadbeef)  // float32 with value 3735928576 (rounded up)
    c =  float64(deadbeef) // float64 with value 3735928559 (exact)
    d =  int32(deadbeef)   // compile error: constant overflows int32
    e =  float64(1e309)    // compile error: constant overflows float64
    f =  uint(-1)          //compile error: constant  underflows uint
)
```

في إعلان المتغير الذي بدون نوع صريح (وهذا يتضمن إعلانات المتغير القصيرة)، تحدد نكهة الثابت الذي بدون نوع النوع الاعتيادي للمتغير ضمنيًا، كما هو موضح في الأمثلة التالية:

```
i :=0      // untyped integer;        implicit int(0)
r :='\000'// untyped rune; implicit   rune('\000')
f :=0.0   // untyped floating-point; implicit float64(0.0)
c :=0i    // untyped complex;         implicit complex128(0i)
```

لاحظ عدم التماثل: تُحوّل الأعداد الصريحة التي دون نوع إلى int، والتي حجمها غير مضمون، ولكن أرقام الفاصلة العائمة التي بدون نوع والأرقام المعقدة/المركبة التي بدون نوع تُحوّل إلى الأحجام ذات الحجم الصريح float64 و complex128. لا تحتوي اللغة على أنواع float و complex دون حجم مناظرة لـ int التي بدون بحجم، ومن الصعب جدًا كتابة خوارزميات صحيحة رقميًا بدون معرفة حجم أنواع بيانات الفاصلة العائمة الخاصة بها.

لو أردنا منح المتغير نوع مختلف، يجب أن نحول الثابت الذي دون نوع بشكل صريح إلى النوع المرغوبة أو توضيح النوع المرغوب في إعلان المتغير، كما هو موضح في الأمثلة التالية:

```
var i = int8(0)
var i int8 = 0
```

إن هذه الأوضاع الاعتيادية مهمة بشكل خاص عند تحويل ثابت بدون نوع إلى قيمة واجهة (انظر الفصل السابع)، حيث أنها تحدد نوعه الديناميكي.

```
fmt.Printf("%T\n", 0)           // "int"
fmt.Printf("%T\n", 0.0)    // "float64"
fmt.Printf("%T\n", 0i)     // "complex128"
fmt.Printf("%T\n", '\000') // "int32" (rune)
```

لقد غطينا الآن أنواع البيانات الأساسية في Go، والخطوة التالية هي توضيح كيف يمكننا دمجها في مجموعات أكبر مثل المصفوفات والبنيات، ثم إلى بنيات بيانات لحل مشكلات البرمجة الأكبر، وهذا موضوع الفصل الرابع.

4- الأنواع المُركّبة – Composite Types

ناقشنا في الفصل الثالث الأنواع الأولية التي تمثل حجر الأساس في تكوين بنيات المعطيات في لغة Go؛ هذه الأنواع الأولية هي كالذرات في عالمنا. سنتطلع في هذا الفصل على الأنواع المركبة (composite types)، وهي الجزيئات التي تتشكل من تجمع الأنواع الأولية بطرق متنوعة. سنتحدث عن أربع أنواع مركبة -المصفوفات، الشرائح، والخرائط، والبنيات (structs)، وسنرى في نهاية الفصل طريقة ترميز البيانات المخزنة في هذه البنيات بشكل JSON وإعادة تحليلها، وطريقة استخدامها لتوليد HTML باستخدام القوالب.

المصفوفات والبنيات هي أنواع تجميعية (aggregate)، أي أنها عبارة عن تجميع لعدد من القيم الأخرى في الذاكرة. المصفوفات متجانسة -أي أن جميع عناصرها من النوع نفسه- بينما البنيات غير متجانسة. أحجام المصفوفات والبنيات ثابتة. بينما الشرائح والخرائط ديناميكية ويزداد حجمها مع زيادة عدد العناصر المضافة إليها.

4.1 المصفوفات - Arrays

المصفوفة هي سلسلة محددة الحجم تتألف من صفر عنصر أو أكثر من نوع محدد. نادرًا ما تستخدم المصفوفات في جو بسبب حجمها الثابت. أما الشريحة فهي تناسب حالات أكثر لأنها يمكن أن تكبر أو تتقلص، لكن علينا نفهم المصفوفات أولًا قبل أن نفهم الشرائح.

يتم الوصول إلى العناصر المفردة في المصفوفة باستخدام تدوين subscript التقليدي، حيث تتراوح قيمة الرقم من صفر وحتى طول المصفوفة ناقص واحد. تعطي الوظيفة len المضمنة في اللغة عدد العناصر في المصفوفة.

```
var a [3]int                // array of 3 integers
fmt.Println(a[0])           // print the first element
fmt.Println(a[len(a)-1])    // print the last element, a[2]
// Print the indices and elements.
for i, v := range a {
    fmt.Printf("%d %d\n", i, v)
}
// Print the elements only.
for _, v := range a {
    fmt.Printf("%d\n", v)
}
```

في الحالة المبدئية، تُهيّأ عناصر المصفوفات الجديدة بالقيمة الصفرية الموافقة لنوع العنصر، وهي الرقم 0 بالنسبة للأنواع الرقمية. يمكننا استخدام مصفوفة حرفية (array literal) لتهيئة عناصر المصفوفة بمجموعة من القيم:

```
var q [3]int = [3]int{1, 2, 3}
var r [3]int = [3]int{1, 2}
fmt.Println(r[2]) // "0"
```

إذا استخدمت ثلاث نقاط "..." في المصفوفة الحرفية بدلًا من تحديد طولها، سوف يحدد الطول اعتمادًا على عدد القيم الأولية المكتوبة. يمكن تبسيط تعريف المصفوفة q كما يلي:

```
q := [...]int{1, 2, 3}
fmt.Printf("%T\n", q) // "[3]int"
```

يعتبر حجم المصفوفة جزءًا من نوعها، ولذلك يعتبر [3]int نوعًا مختلفًا عن [4]int. يجب أن يكون الطول تعبيرًا ثابتًا، أي يمكن حساب الناتج النهائي وقت ترجمة البرنامج.

```
q := [3]int{1, 2, 3}
q = [4]int{1, 2, 3, 4} // compile error: cannot assign [4]int to [3]int
```

كما سنرى فيما بعد، صيغة تعريف المصفوفة الحرفية تتشابه مع الشرائح والخرائط والبنيات. الصيغة المبينة أعلاه تعرف سلسلة من القيم المرتبة، لكن يمكن أيضًا تعريف سلسلة من أزواج (فهرس-قيمة)، كالتالي:

```
type Currency int
const (
    USD Currency = iota
    EUR
    GBP
    RMB
)
symbol := [...]string{USD: "$", EUR: "€", GBP: "£", RMB: "¥"}
fmt.Println(RMB, symbol[RMB]) // "3 ¥"
```

في هذه الصيغة، تظهر الفهارس بأي ترتيب كان، ويمكن إغفال بعضًا منها؛ وستأخذ العناصر المغلقة قيمًا صفرية تناسب نوع العنصر. مثلًا،

```
r := [...]int{99: -1}
```

تُعرّف التعليمة السابقة المصفوفة r التي تحوي 100 عنصر، كلها أصفار عدا الأخير، الذي يأخذ القيمة 1-.

إذا كان نوع عناصر المصفوفة قابلًا للمقارنة، كان نوع المصفوفة قابلًا للمقارنة أيضًا، بحيث يمكننا مقارنة مصفوفتين من ذلك النوع باستخدام العامل == الذي يبين ما إذا كانت جميع العناصر من المصفوفتين متساوية أم لا. أما العامل != فيعطي نفي تلك النتيجة.

```
a := [2]int{1, 2}
b := [...]int{1, 2}
c := [2]int{1, 3}
fmt.Println(a == b, a == c, b == c) // "true false false"
d := [3]int{1, 2}
fmt.Println(a == d) // compile error: cannot compare [2]int == [3]int
```

كمثال آخر، تنتج الدالة Sum256 في الحزمة crypto/sha256 قيمة التلبيد (hash) التشفيري SHA256 أو خلاصة (digest) رسالة ما مخزنة في شريحة بايتات عشوائية. تتألف الخلاصة من 256 خانة، ولذلك نوعها هو[32]byte. إذا كانت الخلاصتان متطابقتان، فمن المرجح جدًا أن الرسالتين متطابقتان أيضًا؛ أما إذا اختلفت الخلاصتان، فالرسالتان مختلفتان. يطبع البرنامج التالي خلاصات SHA256 للرسالتين "x" و"X" ويقارن بينهما:

```
gopl.io/ch4/sha256
import "crypto/sha256"
func main() {
    c1 := sha256.Sum256([]byte("x"))
    c2 := sha256.Sum256([]byte("X"))
    fmt.Printf("%x\n%x\n%t\n%T\n", c1, c2, c1 == c2, c1)
    // Output:
    // 2d711642b726b04401627ca9fbac32f5c8530fb1903cc4db02258717921a4881
    // 4b68ab3847feda7d6c62c1fbcbeebfa35eab7351ed5e78f4ddadea5df64b8015
    // false
    // [32]uint8
}
```

يختلف المُدخَل الأول عن الثاني ببت واحد فقط، لكن نصف بتات الخلاصتين مختلفة عن بعضها تقريبًا. لاحظ أفعال Printf: استخدمنا x% لطباعة كافة عناصر مصفوفة أو شريحة بايتات بالتمثيل الست عشري، واستخدمنا t% لإظهار قيمة منطقية (بوليان)، وT% لعرض نوع القيمة.

عند استدعاء وظيفة، تسند نسخة من كل قيمة من قيم المعطيات إلى متغير المعامل المقابل له، أي أن الوظيفة تستقبل نسخة عن القيمة، وليس المتغير الأصلي نفسه. تمرير المصفوفات الكبيرة بهذه الطريقة قد يكون غير فعال، وأي تغييرات قد تجريها الوظيفة على عناصر المصفوفة ستؤثر على النسخة فقط، وليس على المصفوفة الأصل. في هذا النطاق، تتعامل Go مع المصفوفات كما تتعامل مع الأنواع الأخرى، لكن هذا السلوك مختلف عن اللغات الأخرى التي تمرر المصفوفات عبرالمرجع (by reference) ضمنيًا.

طبعًا، يمكننا تمرير مؤشر إلى المصفوفة بشكل صريح حتى تظهر كافة التغييرات التي تجريها الوظيفة على عناصر المصفوفة على المصفوفة الأصل. مثلًا، الدالة التالية تصفر محتويات مصفوفة من النوع [32]byte:

```
func zero(ptr *[32]byte) {
     for i := range ptr {
       ptr[i] = 0
     }
```

```
}
```

المصفوفة الحرفية {}byte[32] تعطي مصفوفة من 32 بايتًا. كل عنصر فيها يملك قيمة صفرية للنوع byte، وهي الصفر. يمكننا استخدام هذه المعلومة لكتابة نسخة أخرى من الدالة zero:

```
func zero(ptr *[32]byte) {
     *ptr = [32]byte{}
}
```

استخدام مؤشر إلى المصفوفة فعال ويسمح للدالة المستدعاة بأن تعدل المتغير الأصلي، لكن رغم ذلك تبقى المصفوفات غير مرنة بسبب حجمها الثابت. لن تقبل الدالة zero مؤشرًا لمتغير من النوع byte[16] مثلًا، كما لا توجد وسيلة لإضافة أو إزالة عناصر من المصفوفات. لهذه الأسباب، نادرًا ما تستخدم المصفوفات كمتغيرات في الوظائف إلا في حالات خاصة مثل خلاصات SHA256 ذات الحجم الثابت؛ بل تستخدم الشرائح بدلًا منها.

تمرين 4.1: اكتب دالة تحصي عدد البتات المختلفة بين خلاصتي SHA256. (انظر PopCount من القسم 2.6.2).

تمرين 4.2: اكتب برنامجًا يطبع خلاصة SHA256 للقيمة الممررة له عبر الدخل القياسي افتراضيًا لكن مع دعم خيار لطباعة خلاصات SHA384 أو SHA512 بدلًا منها.

4.2 الشرائح - Slices

تمثل الشرائح سلاسل متغيرة الطول من عناصر لها النوع نفسه. يكتب نوع الشريحة على شكل T[]، حيث عناصر الشريحة من النوع T؛ أي أنها مثل المصفوفة ولكن دون تحديد الحجم.

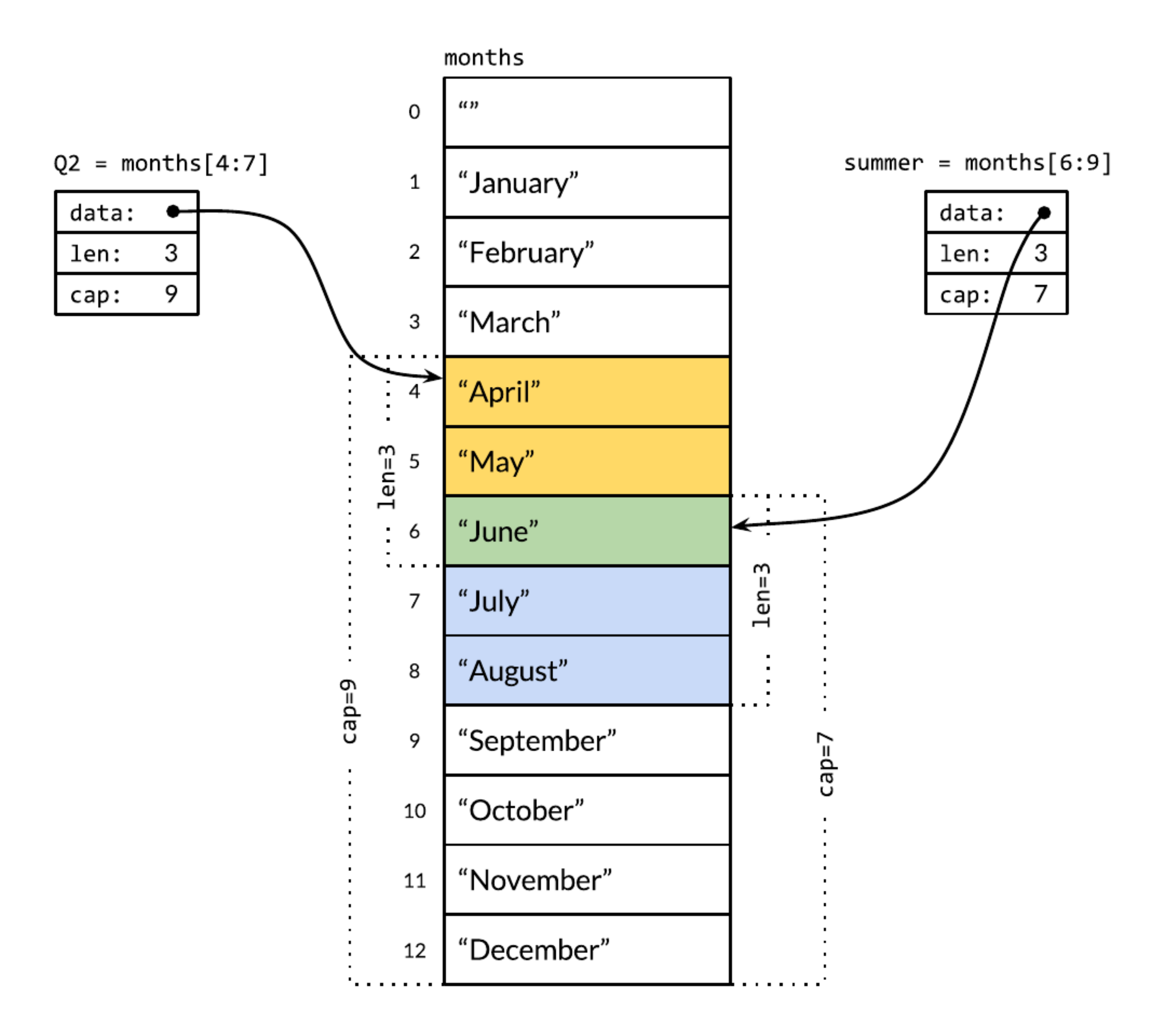
الشرائح والمصفوفات مترابطتان جدًا. فالشريحة هي بنية بيانات تسمح بالوصول إلى سلسلة جزئية من عناصر مصفوفة أو لجميع عناصرها، وهو ما يعرف بالمصفوفة الداخلية (underlying array). تتألف الشريحة من ثلاث مكونات: **مؤشر**، و**طول**، و**سعة**. يؤشر المؤشر إلى العنصر الأول من المصفوفة التي تتيح الشريحة الوصول إليه، ولا يشترط أن يكون العنصر الأول في المصفوفة. الطول هو عدد عناصر الشريحة؛ ولا يمكن أن يتجاوز السعة، التي تمثل عادة عدد العناصر بين بداية الشريحة ونهاية المصفوفة الداخلية. تعطي الوظائف المضمنة len و cap هذه القيم.

يمكن أن تشترك عدة شرائح في مصفوفة داخلية واحدة وقد تشير إلى أجزاء متداخلة منها. يظهر الشكل 4.1 مصفوفة من السلاسل الحرفية لأسماء شهور السنة، وشريحتين متداخلتين منها. عرفت المصفوفة كالتالي:

```
months := [...]string{1: "January", /* ... */, 12: "December"}
```

بحيث يناير هو العنصر months[1] وديسمبر هو months[12]. في الحالة العادية، يستخدم العنصر رقم 0 لتخزين القيمة الأولى، لكن بما أن الشهور ترقم دائمًا من 1، يمكننا إغفال ذلك العنصر وسيتم تهيئته تلقائيًا بسلسلة محرفية فارغة.

ينشئ عامل التشريح s[i:j]، حيث 0 ≤ i ≤ j ≤ cap(s)، شريحة جديدة تشير إلى العناصر من i وحتى j-1 من السلسلة s، التي قد تكون مصفوفة، أو مؤشرًا إلى مصفوفة، أو شريحة أخرى. تحوي الشريحة الناتجة j-i عنصرًا. إذا أغفلت قيمة i فسوف تعتبر 0، أما إذا أغفلت قيمة j فسوف تعتبر len(s). وبالتالي، تشير الشريحة months[1:13] إلى كافة الشهور المقبولة، وكذلك الشريحة months[1:]؛ أما الشريحة months[:] فتشير إلى المصفوفة كاملةً. دعنا نعرف شريحتين متداخلتين للربع الثاني من السنة ولشهور الصيف الشمالي:



الشكل 4.1. شريحتان متراكبتان من مصفوفة الشهور

```
Q2 := months[4:7]
summer := months[6:9]
fmt.Println(Q2)      // ["April" "May" "June"]
fmt.Println(summer)   // ["June" "July" "August"]
```

شهر يونيو مضمن في الشريحتين وهو الناتج الوحيد عن الاختبار التالي (ضعيف الفاعلية) للعناصر المشتركة:

```
for _, s := range summer {
    for _, q := range Q2 {
        if s == q {
            fmt.Printf("%s appears in both\n", s)
        }
    }
}
```

التشريح إلى قيمة أكبر من cap(s) يسبب هلعًا، لكن التشريح إلى قيمة أكبر من len(s) يمدد الشريحة، أي قد تكون النتيجة أطول من الأصل:

```
fmt.Println(summer[:20]) // panic: out of range
endlessSummer := summer[:5] // extend a slice (within capacity)
fmt.Println(endlessSummer) // "[June July August September October]"
```

لاحظ التشابه بين عملية تقطيع السلاسل النصية إلى سلاسل جزئية وبين معامل التشريح على شرائح []byte. كلا العمليتين تكتب x[m:n]، وكلاهما ينتج سلسلة جزئية من البايتات الأصلية، وكلا العمليتان تأخذ زمنًا ثابتًا. ينتج التعبير x[m:n] سلسلة نصية إذا كان x سلسلة نصية، أو يعطي []byte إذا كان x من النوع []byte.

وبما أن الشريحة تحوي مؤشرًا لعنصر من مصفوفة، فإن تمرير الشريحة إلى وظيفة يسمح لها بالتعديل على عناصر المصفوفة الداخلية. بكلمات أخرى، نسخ الشريحة يعطي اسمًا مستعارًا (alias) (انظر 2.3.2) للمصفوفة الداخلية.

تعكس الدالة reverse عناصر شريحة []int في أماكنها، ويمكن تطبيقها على الشرائح بأي طول كان.

```
gopl.io/ch4/rev
// reverse reverses a slice of ints in place.
func reverse(s []int) {
    for i, j := 0, len(s)-1; i < j; i, j = i+1, j-1 {
        s[i], s[j] = s[j], s[i]
    }
}
```

هنا نعكس المصفوفة كلها:

```
a := [...]int{0, 1, 2, 3, 4, 5}
reverse(a[:])
fmt.Println(a) // "[5 4 3 2 1 0]"
```

من الطرق البسيطة لتدوير (rotate) الشريحة إلى اليسار بمقدار n عنصر هي تطبيق وظيفة reverse ثلاث مرات، أول مرة على n من العناصر من بداية الشريحة، والمرة الثانية على بقية عناصر الشريحة، وأخيرًا على الشريحة بكاملها. (للتدوير نحو اليمين، اجعل الاستدعاء الثالث في البداية.)

```
s := []int{0, 1, 2, 3, 4, 5}
// Rotate s left by two positions.
reverse(s[:2])
reverse(s[2:])
reverse(s)
fmt.Println(s) // "[2 3 4 5 0 1]"
```

لاحظ اختلاف التعبير الذي يهيئ الشريحة s عن التعبير الذي استخدم لتهيئة المصفوفة a. الشريحة الحرفية (slice literal) تبدو مثل مصفوفة حرفية، سلسلة من القيم التي تفصلها فواصل ويحصرها قوسان معقوفان، لكن الحجم لا يعطى. هذه التعليمة تنشئ مصفوفة ضمنيًا بالحجم المناسب وتولد شريحة تشير إليها. وكما هي المصفوفات الحرفية، يمكن أن تحدد القيم في الشرائح الحرفية بالترتيب، أو يمكن أن تعطى فهارسها بشكل صريح، أو يمكن استخدام مزيج من الأسلوبين.

بخلاف المصفوفات، لا يمكن مقارنة الشرائح، فلا يمكننا استخدام العامل == لاختبار هل تحوي شريحتين العناصر نفسها. توفر المكتبة القياسية وظيفة bytes.Equal المحسنة جدًا (highly optimized) للمقارنة بين شريحتين من البايتات (نوع []byte)، لكن الأنواع الأخرى من الشرائح تحتاج أن نقارن بينها بأنفسنا:

```go
func equal(x, y []string) bool {
    if len(x) != len(y) {
        return false
    }
    for i := range x {
        if x[i] != y[i] {
            return false
        }
    }
    return true
}
```

بما أن هذه اختبار التساوي "العميق" طبيعي، وأنه لا يكلف أكثر من الناحية التنفيذية من عامل == الذي يستخدم مع مصفوفات السلاسل النصية، فقد يحيرك أن مقارنات الشرائح لا تعمل أيضًا بهذه الطريقة. هناك مشكلتان يعاني منهما هذا الاختبار العميق. أولًا، عناصر الشرائح غير مباشرة، بعكس عناصر المصفوفات، وهذا يسمح للشريحة بأن تحوي نفسها. ورغم أن هناك طرقًا للتعامل مع مثل هذه الحالات، إلا أنه لا توجد طريقة بسيطة، وفعالة، والأهم من هذا لا توجد طريقة واضحة.

ثانيًا، بما أن عناصر الشرائح غير مباشرة، قد تحوي الشريحة عناصر مختلفة في الأوقات المختلفة عندما تتغير محتويات المصفوفة الداخلية. وبما أن جداول التلبيد مثل نوع خريطة في لغة Go تأخذ نسخًا سطحية فقط عن مفاتيحها، فإن هذا يفرض أن المساواة بين المفاتيح أن تُبقى نفسها طوال حياة جدول التلبيد. وبالتالي ستجعل المساواة العميقة الشرائح غير ملائمة للاستخدام كمفاتيح في الخرائط. بالنسبة للأنواع المرجعية مثل المؤشرات والقنوات، يختبر عامل == مساواة المرجع (reference identity)، أي أنه يختبر هل تشير القيمتان إلى الشيء نفسه أم لا. واختبار المساواة مع الشرائح بأسلوب "سطحي" قد يكون مفيدًا، وسيحل المشكلة مع الخرائط، لكن اختلاف سلوك العامل == بين الشرائح والمصفوفات قد يكون مربكًا. الخيار الأكثر أمانًا هو منع مقارنة الشرائح من الأصل.

المقارنة الوحيدة المسموحة مع الشرائح هي المقارنة مع القيمة nil، كما يلي

```
if summer == nil { /* ... */ }
```

القيمة الصفرية للشرائح هي nil. والشريحة ذات القيمة الصفرية nil لا تملك مصفوفة داخلية. كما أن طولها وسعتها تساويان الصفر، لكن هناك شرائح أخرى ليست nil وطولها وسعتها تساوي الصفر أيضًا، مثل {}int[] أو make([]int, 3)[3:]. ومثل بقية الأنواع التي قد تأخذ القيمة nil، يمكن كتابة قيمة nil لأي نوع من الشرائح باستخدام تعبير تحويل مثل []int(nil).

```
var s []int    // len(s) == 0, s == nil
s = nil        // len(s) == 0, s == nil
s = []int(nil)   // len(s) == 0, s == nil
s = []int{}    // len(s) == 0, s != nil
```

فإذا كنت تحتاج إلى اختبار الشريحة لتعرف إذا كانت فارغة، استخدم len(s) == 0، ولا تستخدم s == nil. ولا تختلف الشرائح الصفرية عن غيرها من الشرائح فيما عدا أنها مساوية للقيمة nil في عمليات المقارنة، أما في العمليات الأخرى فشأنها شأن أي شريحة طولها صفر؛ واستدعاء reverse(nil) مثلًا مسموح تمامًا. يجب أن تعمل دوال Go مع كافة الشرائح صفرية الطول بالطريقة نفسها ما لم يذكر خلاف ذلك بشكل واضح، سواء أكانت قيمة هذه الشرائح nil أم لم تكن.

تولد الدالة المضمنة make شريحة من نوع عناصر وطول وسعة محددين. يمكن إغفال قيمة السعة، وفي تلك الحالة تعتبر السعة مساوية للطول.

```
make([]T, len)
make([]T, len, cap)   // same as make([]T, cap)[:len]
```

تحت الستار، تولد make مصفوفة دون اسم وتعيد شريحة منها؛ فلا يمكن الوصول للمصفوفة إلا من خلال الشريحة. في الصيغة الأولى، تظهر الشريحة كافة عناصر المصفوفة. أما في الصيغة الثانية، فتعرض الشريحة عدد len عنصر من بداية المصفوفة فقط، لكن سعة الشريحة تشمل المصفوفة كاملةً. أما العناصر الإضافية فهي متاحة للتوسع مستقبلًا.

4.2.1 دالة الإلحاق append

تستخدم دالة المضمنة append لإضافة عناصر إلى الشرائح:

```
var runes []rune
for _, r := range "Hello, 世界" {
    runes = append(runes, r)
}
fmt.Printf("%q\n", runes) // "['H' 'e' 'l' 'l' 'o' ',' ' ' '世' '界']"
```

تستخدم الحلقة دالة append لبناء شريحة من تسع حروف من السلسلة النصية، رغم أن هذه المشكلة يمكن حلها بطريقة أنسب باستخدام التحويل المضمن في أصل اللغة []rune("Hello, 世界")

وظيفة append لازمة لفهم طريقة عمل الشرائح، لذلك دعنا نلق نظرة على ما يجري. فيما يلي نسخة تدعى appendInt مخصصة لشرائح []int:

```
gopl.io/ch4/append
func appendInt(x []int, y int) []int {
    var z []int
    zlen := len(x) + 1
    if zlen <= cap(x) {
        // There is room to grow.  Extend the slice.
        z = x[:zlen]
    } else {
        // There is insufficient space.  Allocate a new array.
        // Grow by doubling, for amortized linear complexity.
        zcap := zlen
        if zcap < 2*len(x) {
            zcap = 2 * len(x)
        }
        z = make([]int, zlen, zcap)
        copy(z, x) // a built-in function; see text
    }
    z[len(x)] = y
    return z
}
```

كل استدعاء للوظيفة appendInt يجب أن يتحقق أن الشريحة تملك سعة كافية لتخزين العناصر الجديدة في المصفوفة الحالية. إذا كانت السعة تكفي فإن الشريحة تمدد عن طريق تعريف شريحة أكبر (ضمن حدود المصفوفة الأصلية)، ونسخ العنصر y إلى المساحة الجديدة، وإعادة الشريحة كنتيجة. تشترك شريحة الدخل x وشريحة الخرج z بالمصفوفة الداخلية نفسها.

أما إذا لم تكن السعة تكفي للتمدد، يجب أن تخصص مصفوفة جديدة أكبر حتى تستوعب النتيجة، ثم تنسخ القيم من الشريحة x إليها، ثم يضاف العنصر الجديد y. وفي هذه الحالة ستشير الشريحة الناتجة z إلى مصفوفة داخلية مختلفة عن المصفوفة التي تشير إليها x.

قد يكون استخدام الحلقات لنسخ العناصر واضحًا، لكن الأسهل استخدام الوظيفة المضمنة copy، التي تنسخ العناصر من شريحة إلى أخرى من النوع نفسه. متغيراها الأول هو وجهة النسخ أما الثاني فهو مصدر النسخ، حيث تحاكي ترتيب المعاملات في تعليمة الإسناد dst = src. قد تشير الشرائح إلى المصفوفة الداخلية نفسها؛ وقد تتداخلان فيما بينهما أيضًا. كما تعيد الوظيفة copy عدد العناصر المنسوخة فعليًا رغم أننا لا نستخدم هذه القيمة هنا، وهو يساوي طول الشريحة الأقصر بين الشريحتين المعطاتين، ولذلك لا يوجد خطر تجاوز الحد أو كتابة شيء خارج المجال.

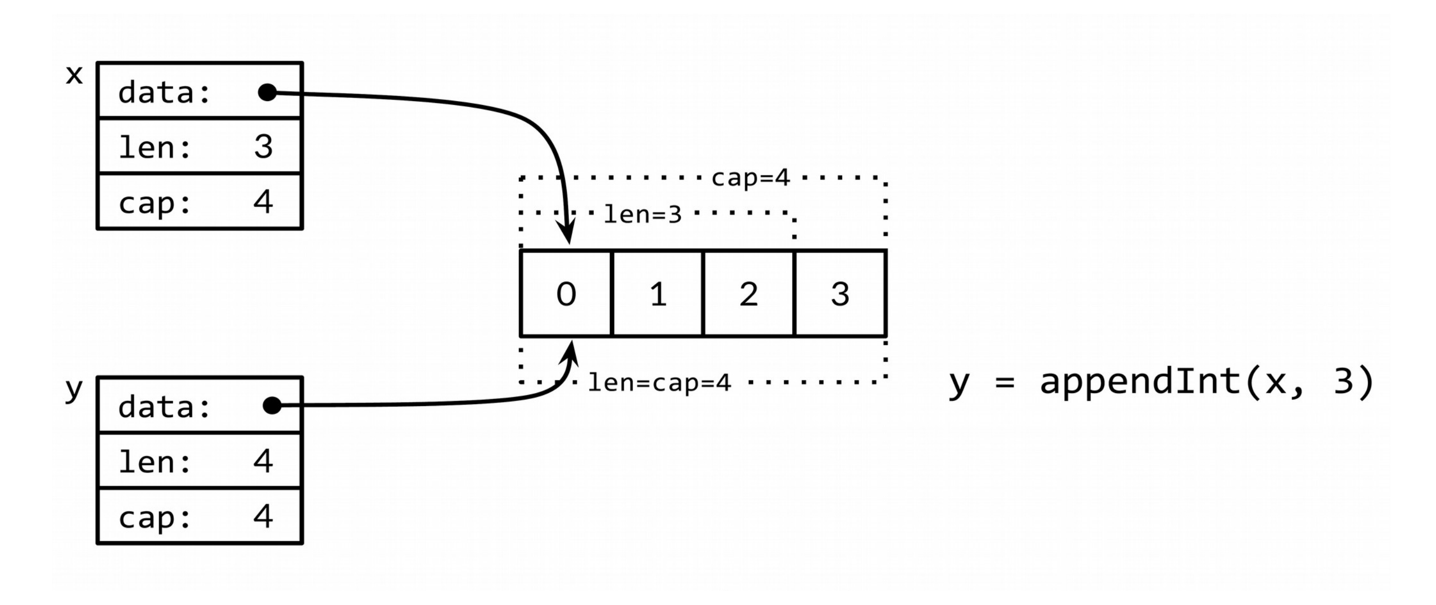
تكون المصفوفة الجديدة أكبر نوعًا ما من الحجم الأدنى اللازم لاستيعاب x و y بهدف زيادة الفاعلية. توسيع المصفوفة إلى ضعف حجمها الأولي في كل مرة نحتاج فيها إلى توسيعها يقلل عدد مرات التوسيع وإعادة التخصيص ويضمن تنفيذ عملية إضافة العناصر في زمن ثابت بشكل وسطي. يبين البرنامج التالي هذا الأثر:

```
func main() {
    var x, y []int
    for i := 0; i < 10; i++ {
        y = appendInt(x, i)
        fmt.Printf("%d  cap=%d\t%v\n", i, cap(y), y)
        x = y
    }
}
```

كل تغيير في السعة يعني عملية إعادة تخصيص مع نسخ:

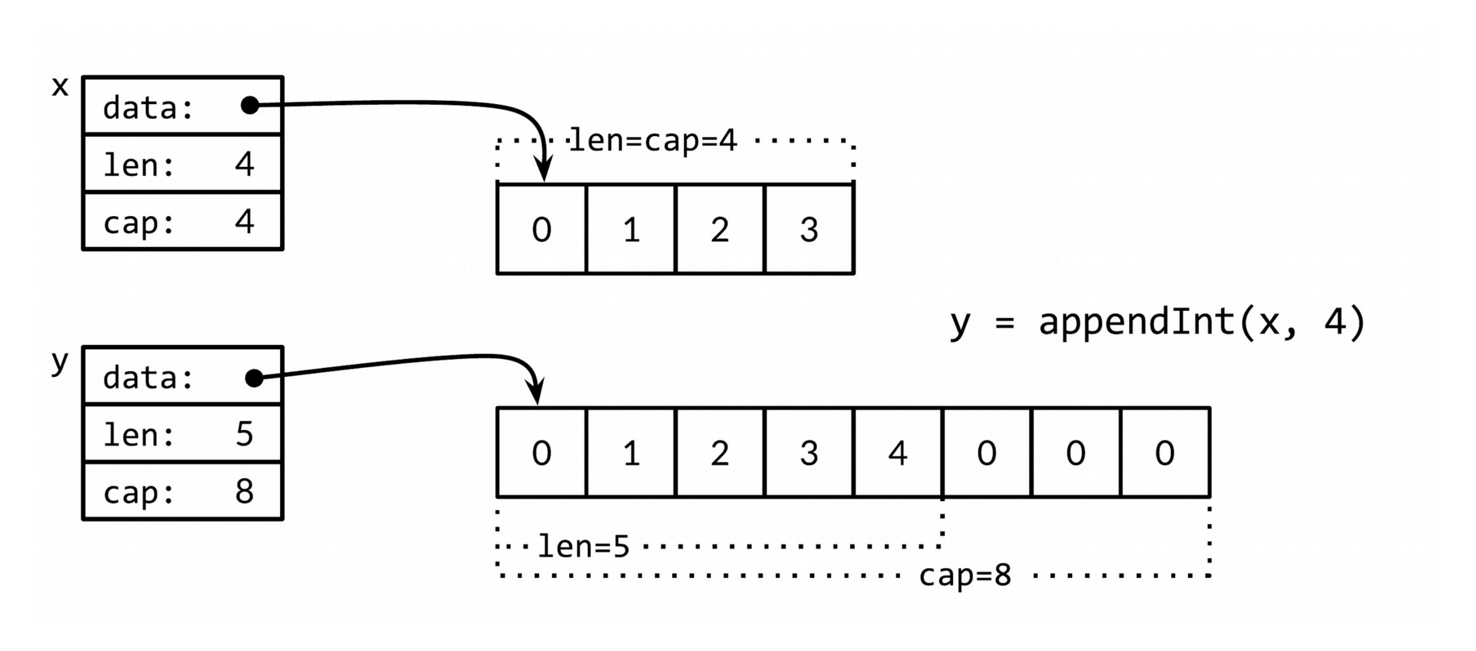
```
0  cap=1    [0]
1  cap=2    [0 1]
2  cap=4    [0 1 2]
3  cap=4    [0 1 2 3]
4  cap=8    [0 1 2 3 4]
5  cap=8    [0 1 2 3 4 5]
6  cap=8    [0 1 2 3 4 5 6]
7  cap=8    [0 1 2 3 4 5 6 7]
8  cap=16   [0 1 2 3 4 5 6 7 8]
9  cap=16   [0 1 2 3 4 5 6 7 8 9]
```

دعنا نلق نظرة أقرب على الخطوة i=3. تحوي الشريحة x ثلاث عناصر [0 1 2] لكن سعتها تساوي 4، أي يوجد عنصر واحد متاح في النهاية، وتستطيع الوظيفة appendInt المتابعة دون إعادة التخصيص عند إضافة العنصر 3. الشريحة الناتجة y تملك طولًا وسعةً يساويان 4، كما تملك المصفوفة الداخلية نفسها التي تملكها الشريحة الأصلية x، كما يوضح الشكل 4.2.



الشكل 4.2: تطبيق وظيفة Append بوجود مساحة للنمو.

في الدورة التالية عندما i=4، لا يوجد فراغ في النهاية، ولذلك تخصص الوظيفة appendInt مصفوفةً جديدة حجمها 8، وتنسخ العناصر الأربعة [0 1 2 3] من x، وتضيف إليهم 4، ألا وهي قيمة i. طول الشريحة الناتجة y يساوي 5 لكن سعتها تساوي 8 ؛ وستتيح الفراغات الثلاثة الناتجة عن هذه الدورة إضافة العناصر الثلاثة التالية دون الحاجة لإعادة تخصيص مصفوفة جديدة في كل مرة. ترتبط كل من الشريحة x والشريحة y بمصفوفة مختلفة هذه المرة. هذه العملية موضحة في الشكل 4.3.



الشكل 4.3: تطبيق وظيفة Append دون مساحة للنمو.

الوظيفة المضمنة append تستخدم استراتيجية توسيع معقدة أكثر من الاستراتيجية المبسطة المستخدمة في appendInt. لا نعرف عادة إذا كان استدعاء append سيسبب عملية إعادة تخصيص، ولذلك لا يمكننا أن نفترض أن

الشريحة الأصلية ستشير إلى نفس المصفوفة التي تشير إليها الشريحة الناتجة، كما لا يمكننا أن نفترض أنها ستشير إلى مصفوفة جديدة. وكذلك لا يجب أن نفترض أن العمليات على عناصر الشريحة القديمة ستؤثر (أو لن تؤثر) على العناصر في الشريحة الجديدة. نتيجة لذلك، من المعتاد أن يتم إسناد نتيجة استدعاء وظيفة append إلى نفس المتغير الذي يحوي قيمة الشريحة التي مررناها كمتغير للوظيفة:

```
runes = append(runes, r)
```

تحديث متغير الشريحة لا يلزم مع وظيفة append وحدها، بل هو لازم مع كل استدعاء لأي وظيفة يحتمل أن تعدل طول الشريحة أو سعتها أو تجعلها تشير إلى مصفوفة داخلية مختلفة. رغم أن عناصر المصفوفة الداخلية غير مباشرة في الشرائح إلا أن طول الشريحة وسعتها ومؤشرها إلى المصفوفة الداخلية هي خصائص مباشرة، ومن المهم أن تتذكر هذا دومًا حتى تستخدم الشرائح بطريقة صحيحة. يجب استخدام عملية إسناد كما في التعليمة السابقة لتحديث هذه القيم. من هذه الناحية، تعتبر الشرائح أنواع مؤشرات "غير خالصة" لكنها تحاكي الأنواع التجميعية مثل هذا البنية:

```
type IntSlice struct {
     ptr *int
     len, cap int
}
```

تضيف وظيفة appendInt عنصرًا وحيدًا إلى الشريحة، لكن الوظيفة المضمنة append تسمح لنا بإضافة عدة عناصر معًا، أو إضافة شريحة كاملة إلى شريحة أخرى:

```
var x []int
x = append(x, 1)
x = append(x, 2, 3)
x = append(x, 4, 5, 6)
x = append(x, x...) // append the slice x
fmt.Println(x)      // "[1 2 3 4 5 6 1 2 3 4 5 6]"
```

يمكننا تعديل appendInt بشكل بسيط لمحاكاة سلوك الوظيفة المضمنة append كما هو مبين أدناه. النقاط الثلاثة "..." في تعريف الوظيفة تجعلها variadic: أي أنها تقبل أي عدد من القيم. وقد استخدمنا النقاط الثلاثة أعلاه عند استدعاء append حتى نعطيها كافة القيم المخزنة في الشريحة. سوف نشرح هذه الآلية بالتفصيل في القسم 5.7.

```
func appendInt(x []int, y ...int) []int {
     var z []int
     zlen := len(x) + len(y)
     // ...expand z to at least zlen...
     copy(z[len(x):], y)
     return z
}
```

المنطق المعتمد لتوسعة المصفوفة الداخلية للشريحة z لم يتغير، وهو غير ظاهر في المثال الأخير.

4.2.2 التقنيات الموضعية في معالجة الشرائح

دعنا نعرض المزيد من الأمثلة عن وظائف تعدل على عناصر الشريحة في نفس المكان، مثل rotate و reverse. وظيفة nonempty تعيد السلاسل المحرفية غير الفارغة من مجموعة السلاسل المحرفية التي تأخذها كمتغير:

```
gopl.io/ch4/nonempty
// Nonempty is an example of an in-place slice algorithm.
package main
import "fmt"
// nonempty returns a slice holding only the non-empty strings.
// The underlying array is modified during the call.
func nonempty(strings []string) []string {
    i := 0
    for _, s := range strings {
        if s != "" {
            strings[i] = s
            i++
        }
    }
    return strings[:i]
}
```

الجزء الغامض هو أن شريحة الدخل وشريحة الخرج تشتركان بالمصفوفة الداخلية نفسها. هذا يوفر عبئ تخصيص مصفوفة جديدة، رغم أن هناك جزء من محتويات data سوف يستبدل، كما نرى من نتيجة تعليمة الطباعة الثانية:

```
data := []string{"one", "", "three"}
fmt.Printf("%q\n", nonempty(data))  // `["one" "three"]`
fmt.Printf("%q\n", data)            // `["one" "three" "three"]`
```

لهذا نستدعي الوظيفة بهذا الشكل عادة: data = nonempty(data).

يمكن كتابة وظيفة nonempty باستخدام append أيضًا:

```
func nonempty2(strings []string) []string {
    out := strings[:0] // zero-length slice of original
    for _, s := range strings {
        if s != "" {
            out = append(out, s)
        }
    }
    return out
}
```

أيًا كانت الطريقة التي نستخدمها، إعادة استخدام المصفوفة بهذا الشكل يفرض علينا أن لا يتم توليد أكثر من قيمة خرج واحدة لكل قيمة من قيم الدخل، وهذا ينطبق على العديد من الخوارزميات التي تعمل على تصفية عناصر سلسلة أو جمع العناصر المتجاورة. هذا الاستخدام المعقد هو الحالة الاستثنائية وليس القاعدة، لكنه قد يكون واضحًا، وفعالًا، ومفيدًا في بعض الحالات.

يمكن استخدام الشرائح لإنشاء رصة (stack). إذا كانت لدينا شريحة فارغة مبدئيًا اسمها stack، يمكننا دفع قيمة جديدة إلى نهاية الشريحة باستدعاء append:

```
stack = append(stack, v) // push v
```

قمة الرصة هي العنصر الأخير:

```
top := stack[len(stack)-1] // top of stack
```

ولتقليص الرصة يمكننا سحب العنصر الأخير كما يلي

```
stack = stack[:len(stack)-1] // pop
```

لإزالة عنصر من وسط الشريحة، مع المحافظة على ترتيب العناصر المتبقية، استخدم copy لتحريك العناصر التالية إلى الوراء خطوة واحدة لتعبئة الفراغ:

} func remove(slice []int, i int) []int

copy(slice[i:], slice[i+1:])

return slice[:len(slice)-1]

{

} ()func main

s := []int{5, 6, 7, 8, 9}

"fmt.Println(remove(s, 2)) // "[5 6 8 9]

{

وإذا لم نرغب في الحفاظ على ترتيب العناصر، يكفينا أن ننقل العنصر الأخير إلى مكان الفراغ:

```
func remove(slice []int, i int) []int {
    slice[i] = slice[len(slice)-1]
    return slice[:len(slice)-1]
}
func main() {
    s := []int{5, 6, 7, 8, 9}
    fmt.Println(remove(s, 2)) // "[5 6 9 8]
}
```

تمرين 4.3: اكتب وظيفة reverse بحيث تستخدم مؤشر شريحة بدلًا من مصفوفة.

تمرين 4.4: اكتب نسخة من rotate تنجز عملها بمرور واحد.

تمرين 4.5: اكتب وظيفة موضعية تزيل العناصر المتطابقة والمتجاورة في شريحة من نوع []string.

تمرين 4.6: اكتب وظيفة موضعية تدمج كل مجموعة من محارف الفراغات حسب ترميز يونيكود (انظر unicode.IsSpace) في شريحة []byte ترميزها UTF-8 في فراغ وحيد بترميز ASCII.

تمرين 4.7: عدل على reverse بحيث تعكس المحارف في شريحة []byte تمثل سلسلة UTF-8، موضعيًا. هل يمكنك كتابتها دون تخصيص ذاكرة جديدة؟

4.3 الخرائط - Maps

جداول التلبيد (hash) هي أحد أذكى بنيات المعطيات وأكثرها استخدامًا في مجالات متنوعة. وهي عبارة عن مجموعة من أزواج مفتاح/قيمة، حيث تتمايز المفاتيح عن بعضها ويمكن قراءة القيمة المرتبطة بالمفتاح أو تحديثها أو حذفها بعد عدد ثابت وسطيًا من المقارنات بين المفاتيح، مهما كان حجم جدول التلبيد كبيرًا.

في لغة Go، تشير الخريطة (map) إلى جدول تلبيد، وتكتب الخريطة بالشكل map[K]V، حيث K و V هي أنواع المفاتيح والقيم. تكون كل المفاتيح في الخريطة من نوع واحد، كما تكون كل القيم من نوع واحد أيضًا، لكن لا يشترط أن تكون المفاتيح والقيم من النوع ذاته. يجب أن يقبل نوع المفاتيح المقارنة باستخدام عامل ==، حتى تستطيع الخريطة أن تعرف إذا كانت قيمة مفتاح معطاة موجودة مسبقًا في الجدول لديها. رغم أن أعداد الفاصلة العائمة تقبل المقارنة، إلا أن اختبار التساوي بين الأعداد ذات الفواصل ليس فكرة جيدة، وخصوصًا إذا كان يحتمل ورود القيمة NaN كما شرحنا في الفصل 3. لا توجد قيود على نوع القيم V.

يمكن استخدام الوظيفة المضمنة make لإنشاء خريطة:

```
ages := make(map[string]int) // mapping from strings to ints
```

يمكننا أيضًا استخدام خريطة حرفية (literal map) لإنشاء خريطة جديدة وتهيئتها ببعض أزواج المفاتيح والقيم الأولية:

```
ages := map[string]int{
     "alice": 31,
     "charlie": 34,
}
```

هذه التعليمات تكافئ ما يلي

```
ages := make(map[string]int)
```

```
ages["alice"] = 31
ages["charlie"] = 34
```

أي أن التعليمة المكافئة لإنشاء خريطة جديدة فارغة هي {}map[string]int.

يوصل إلى عناصر الخريطة باستخدام الصيغة المعتادة:

```
ages["alice"] = 32
fmt.Println(ages["alice"]) // "32"
```

وتحذف عبر استدعاء الوظيفة المضمنة delete:

```
delete(ages, "alice") // remove element ages["alice"]
```

كل هذه العمليات آمنة ولو لم يكن العنصر موجودًا في الخريطة؛ فالبحث في الخريطة باستخدام مفتاح يعيد القيمة الصفرية للنوع المناسب، مثلًا، التعليمات التالية ستعمل ولو لم يكن "bob" موجودًا بين المفاتيح في الخريطة لأن قيمة ["ages["bob ستكون 0.

```
ages["bob"] = ages["bob"] + 1 // happy birthday!
```

إن أشكال الإسناد المختصرة x += y و ++x تعمل أيضًا مع عناصر الخريطة، وبالتالي يمكننا إعادة كتابة العبارة أعلاه كالتالي:

```
ages["bob"] += 1
```

أو بدقة أكثر كالتالي:

```
ages["bob"]++
```

ولكن عنصر الخريطة ليس متغيرًا، ولا يمكننا أخذ عنوانه:

```
_ = &ages["bob"]  // compile error: cannot take address of map element
```

أحد أسباب عدم قدرتنا على أخذ عنوان عنصر خريطة هي أن بناء خريطة يمكن أن يسبب إعادة تلبيد (rehashing) للعناصر الموجودة إلى مواقع تخزين جديدة، وبالتالي يصبح العنوان غير صالح.

يمكننا تحديد عدد كل ثنائيات المفتاح/القيمة في الخريطة باستخدام حلقة for قائمة على النطاق مشابهة للتي رأيناها في الشرائح. إن التكرارات المتتابعة للحلقة تجعل متغيرات الاسم والعمر منضبطة على ثنائي المفتاح/القيمة التالي:

```
for name, age := range ages {
    fmt.Printf("%s\t%d\n", name, age)
}
```

إن ترتيب تكرار الخريطة غير محدد، ويمكن للتطبيقات المختلفة استخدام وظيفة hash مختلفة، وهو ما يؤدي لترتيب مختلف. وفي الواقع العملي، سيكون الترتيب عشوائيا، ويتباين من تنفيذ إلى آخر. إن هذا متعمد، ويساعد تباين التسلسل على إجبار البرامج على أن تكون قوية ودقيقة خلال التطبيقات. يجب أن نفرز المفاتيح بشكل صريح من أجل تحديد عدد ثنائيات المفتاح/القيمة بالترتيب، على سبيل المثال، باستخدام وظيفة String من حزمة sort لو كانت المفاتيح سلاسل. هذا نمط شائع:

```
import "sort"
var names []string
for name := range ages {
    names = append(names, name)
}
sort.Strings(names)
for _, name := range names {
    fmt.Printf("%s\t%d\n", name, ages[name])
}
```

نظرًا لأننا نعلم الحجم النهائي للأسماء منذ البداية، سيكون من الكفء أكثر تخصيص مصفوفة بالحجم المطلوب مقدمًا. تصنع العبارة أدناه شريحة فارغة مبدئيًا، ولكنها تمتلك سِعة كافية لحمل كل مفاتيح خريطة ages:

```
names := make([]string, 0, len(ages))
```

سنحتاج في أول حلقة نطاق أعلاه مفاتيح خريطة ages فقط، وبالتالي نحذف متغير الحلقة الثاني. وفي الحلقة الثانية، سنحتاج فقط إلى العناصر الخاصة بشريحة names، وبالتالي سنستخدم المُعرّف الفارغ _ لتجاهل المتغير الأول، أي الفهرس.

إن القيمة الصفرية لنوع الخريطة هي nil، أي مرجع يشير إلى عدم وجود جدول تلبيد على الإطلاق.

```
var ages map[string]int
fmt.Println(ages == nil) // "true"
fmt.Println(len(ages) == 0) // "true"
```

إن معظم العمليات على الخريطة، بما فيها lookup و delete و len وحلقات النطاق، من الآمن أدائها في مرجع خريطة nil، حيث أنه يعمل كخريطة فارغة، ولكن التخزين في خريطة nil يسبب هلع:

```
ages["carol"] = 21 // panic: assignment to entry in nil map
```

يجب أن تخصص الخريطة قبل أن تتمكن من التخزين فيها.

إن دخول عنصر خريطة عن طريق التسجيل (subscripting) ينتج عنه قيمة دائمًا، ولو كان المفتاح موجودًا في الخريطة، فستحصل على قيمة مناظرة، ولو لم تحصل عليها، ستحصل على القيمة الصفرية لنوع العنصر، كما رأينا في

ages["bob"]. لا بأس بهذا في العديد من الحالات، ولكن أحيانًا ستحتاج لمعرفة ما إذا كان العنصر موجودا حقًا أم لا. على سبيل المثال، لو كان نوع العنصر رقميًا، فقد تضطر للتمييز بين العنصر غير الموجود وبين العنصر الذي يمتلك القيمة صفر، باستخدام اختبار كهذا:

```
age, ok := ages["bob"]
if !ok { /* "bob" is not a key in this map; age == 0. */ }
```

سترى هاتين العبارتين مُدمجتين عادة، هكذا:

```
if age, ok := ages["bob"]; !ok { /* ... */ }
```

ينتج عن تسجيل خريطة في هذا السياق قيمتين، الثاني هي قيمة منطقة (بوليان) توضح ما إذا كان العنصر حاضرًا أم لا. ويُطلق على متغير القيمة المنطقية عادة ok، وخاصة لو كان يُستخدم فورًا في حالة if.

كما هو الحال مع الشرائح، لا يمكن مقارنة النتائج مع بعضها، فالمقارنة الممكنة الوحيدة هي المقارنة مع nil. ولاختصار ما إذا كانت الخرائط تحتوي على نفس المفاتيح، ونفس القيم المرتبطة بها أم لا، يجب أن نكتب حلقة:

```
func equal(x, y map[string]int) bool {
    if len(x) != len(y) {
        return false
    }
    for k, xv := range x {
        if yv, ok := y[k]; !ok || yv != xv {
            return false
        }
    }
    return true
}
```

لاحظ كيف نستخدم ok! للتمييز بين الحالات "المفقودة" و"الموجودة ولكن صفرية". لو كنا كتبنا xv != y[k] فإن الاستدعاء أدناه كان سيقدم معطيات متساوية وهذا خاطئ:

```
// True if equal is written incorrectly.
equal(map[string]int{"A": 0}, map[string]int{"B": 42})
```

لا تقدم لغة جو نوع المجموعة set ، حيث أن الخريطة يمكن أن تقوم بهذا الدور نظرًا لكون مفاتيحها مميزة. للتوضيح، يقرأ برنامج dedup تسلسل السطور ويطبع فقط أول حالة وقوع لكل سطر مميز. (وهو تنويعة على برنامج dup الذي قدمناه في القسم 1.3). يستخدم برنامج dedup خريطة تمثل مفاتيحها مجموعة من السطور التي ظهرت بالفعل لضمان عدم طباعة حالات الوقوع التالية.

```
gopl.io/ch4/dedup
func main() {
    seen := make(map[string]bool) // a set of strings
```

```
    input := bufio.NewScanner(os.Stdin)
    for input.Scan() {
        line := input.Text()
        if !seen[line] {
            seen[line] = true
            fmt.Println(line)
        }
    }
    if err := input.Err(); err != nil {
        fmt.Fprintf(os.Stderr, "dedup: %v\n", err)
        os.Exit(1)
    }
}
```

يصف مُبرمجو Go عادة الخريطة المستخدمة بهذه الطريقة بأنها "مجموعة سلاسل" دون كلام كثير، ولكن انتبه، ليست كل قيم map[string]bool مجموعات بسيطة، فبعضها يحتوي على قيم صحيحة وخاطئة معًا.

نحتاج أحيانًا إلى خريطة أو مجموعة مفاتيحها شرائح، ولكن لا يمكن التعبير عنها بشكل مباشر لأن مفاتيح الخريطة يجب أن تكون قابلة للمقارنة. مع ذلك، يمكن فعل هذا في خطوتين، الأولى أن نُعرّف وظيفة المساعِد k التي تخطط كل مفتاح مع سلسلة، مع افتراض أن k(x) == k(y) فقط لو اعتبرنا x و y مكافئين لبعضهما. نصنع بعدها خريطة مفاتيحها سلاسل، ونطبق وظيفة المساعد على كل مفتاح قبل أن ندخل الخريطة.

يستخدم المثال أدناه خريطة لتسجيل عدد مرات استدعاء Add مع قائمة معينة من السلاسل. يستخدم المثال fmt.Sprintf لتحويل شريحة سلاسل إلى سلسلة واحدة يمكن اعتبارها مفتاح خريطة مناسب، ويقتبس كل عنصر شريحة باستخدام %q لتسجيل حدود السلسلة بدقة:

```
var m = make(map[string]int)
func k(list []string) string { return fmt.Sprintf("%q", list) }
func Add(list []string)    { m[k(list)]++ }
func Count(list []string) int { return m[k(list)] }
```

يمكن استخدام نفس الطريقة على أي نوع مفتوح غير قابل للمقارنة، وليس الشرائح فقط، وهو مفيد أيضًا في أنواع المفتاح القابلة للمقارنة عندما تريد تعريف للمساواة بخلاف ==، مثل مقارنات السلاسل غير الحساسة للحالة. لا يجب أن يكون النوع k(x) سلسلة، بل أن أي نوع قابل للمقارنة ذو الخاصية المكافِئة المرغوبة، مثل الأعداد الصحيحة والمصفوفات والبنيات، سيكون مناسب.

إليك مثال آخر على الخرائط أثناء التطبيق، وهو برنامج يحسب مرات وقوع كل نقطة شفرة Unicode مميزة في مدخلاته. نظرًا لوجود عدد كبير من الحروف المحتملة، سنجد أن جزء صغير فقط منها سيظهر في أي مستند محدد، والخريطة هي طريقة طبيعية لتتبع الحروف التي ظهرت والأعداد المناظرة لها.

```
gopl.io/ch4/char count
```

```go
// Charcount computes counts of Unicode characters.
package main
import (
    "bufio"
    "fmt"
    "io"
    "os"
    "unicode"
    "unicode/utf8"
)
func main() {
    counts := make(map[rune]int)    // counts of Unicode characters
    var utflen [utf8.UTFMax + 1]int // count of lengths of UTF-8 encodings
    invalid := 0                    // count of invalid UTF-8 characters
    in := bufio.NewReader(os.Stdin)
    for {
        r, n, err := in.ReadRune() // returns rune, nbytes, error
        if err == io.EOF {
            break
        }
        if err != nil {
            fmt.Fprintf(os.Stderr, "charcount: %v\n", err)
            os.Exit(1)
        }
        if r == unicode.ReplacementChar && n == 1 {
            invalid++
            continue
        }
        counts[r]++
        utflen[n]++
    }
    fmt.Printf("rune\tcount\n")
    for c, n := range counts {
        fmt.Printf("%q\t%d\n", c, n)
    }
    fmt.Print("\nlen\tcount\n")
    for i, n := range utflen {
        if i > 0 {
            fmt.Printf("%d\t%d\n", i, n)
        }
    }
    if invalid > 0 {
        fmt.Printf("\n%d invalid UTF-8 characters\n", invalid)
    }
}
```

تقوم الدالة ReadRune بفك الترميز UTF-8، وتعيد ثلاث قيم: الـ rune المفكوك ترميزه، وطول ترميز UTF-8 الخاص به بالبايتات، وقيمة الخطأ. إن الخطأ الوحيد الذي نتوقعه هو "نهاية الملف". ولو كان المُدخَل ترميز UTF-8 غير قانوني للـ rune، فإن الـ rune المُعاد سيكون unicode.ReplacementChar وطوله 1.

يطبع برنامج charcount أيضًا عدد أطوال ترميزات UTF-8 الخاصة بالـ runes التي تظهر في المُدخَل، والخريطة ليست أفضل هيكل بيانات مناسبة لهذا، حيث أن أطوال الترميز تتراوح فقط من 1 إلى utf8.UTFMax (والذي يمتلك القيمة 4)، بينما تكون المصفوفة مضغوطة أكثر.

قمنا بتشغيل charcount على هذا الكتاب نفسه في وقت ما كتجربة، وبالرغم من أن معظمه باللغة الإنجليزية، إلا أنه يحتوي على عدد لا بأس به من حروف non-ASCII characters. وهذه أبرز 10 منها:

° 27 世 15 界 14 é 13 ˣ 10 ≤ 5 × 5 国 4 � 4 □ 3

وهنا توزيع أطوال كل ترميزات UTF-8:

```
len  count
1    765391
2    60
3    70
4    0
```

إن نوع قيمة الخريطة نفسها يمكن أن يكون نوعا مُركّبا، مثل خريطة أو شريحة. وفي الشفرة التالية، كان نوع مفتاح الرسم البياني هو سلسلة، وقيمة النوع هي map[string]bool، والتي تمثل مجموعة من السلاسل. من الناحية المفاهيمية، يخطط الرسم البياني graph سلسلة ويربطها بمجموعة من السلاسل ذات الصلة، والتي تُعَد خلفاء لها في الرسم البياني المُوجّه:

```
gopl. io/ch4/graph
var graph = make(map[string]map[string]bool)
func addEdge(from, to string) {
    edges := graph[from]
    if edges == nil {
        edges = make(map[string]bool)
        graph[from] = edges
    }
    edges[to] = true
}
func hasEdge(from, to string) bool {
    return graph[from][to]
}
```

توضح وظيفة addEdge الطريقة الاصطلاحية لرسم الخرائط بكسل، أي بدء كل قيمة مع ظهور مفتاحها لأول مرة. توضح وظيفة hasEdge كيف تعمل القيمة الصفرية لمُدخل الخريطة المفقود عادة: حتى لو يكن هناك "from" و"to"، ستقدم graph[from][to] نتيجة ذات معنى دائمًا.

تمرين 4.8: عدّل برنامج charcount ليحسب الحروف والأرقام وغيرها في فئات Unicode، باستخدام وظائف مثل unicode.IsLetter.

تمرين 4.9: اكتب برنامج wordfreq لتقديم تكرار كل كلمة في ملف نصي مُدخل. استدعي input.Split(bufio.ScanWords) قبل أول استدعاء لـ Scan لتقسيم المُدخل إلى كلمات بدلًا من سطور.

4.4 البِنيات - Structs

إن البنية أو struct هي نوع بيانات مُجمّع يجمع القيم ذات الاسم الصفري أو أكثر من الأنواع العشوائية، ويعتبرها كيان منفرد. يُطلق على كل قيمة "حقل" (field)، والمثال الكلاسيكي على البنية في معالجة البيانات هو سجل الموظف، والذي يحتوي على حقول مثل: رقم الهوية فريد، اسم الموظف، العنوان، تاريخ الميلاد، المنصب، الراتب، المدير، إلخ. تُجمع كل هذه الحقول في كيان منفرد يمكن نسخه كوحدة واحدة، وتمريره إلى الوظائف وإعادته بواسطتهم، وتخزينه في مصفوفات، إلخ.

تعلن هاتان العبارتان عن نوع بنية يُسمى الموظف (Employee)، ومتغير يُسمى dilbert، وهو مثال على Employee:

```go
type Employee struct {
    ID        int
    Name      string
    Address   string
    DoB       time.Time
    Position  string
    Salary    int
    ManagerID int
}
var dilbert Employee
```

يوصل للحقول الفردية في dilbert باستخدام تدوين نقطي مثل dilbert.Name و dilbert.DoB. ونظرًا لكون dilbert متغير، فإن حقوله متغيرات أيضًا، وبالتالي يمكننا أن نعيّن للحقل:

```go
dilbert.Salary -= 5000 // demoted, for writing too few lines of code
```

أو نأخذ عنوانه ونصل إليه عبر مؤشر:

```go
position := &dilbert.Position
*position = "Senior " + *position // promoted, for outsourcing to Elbonia
```

يعمل التدوين النقطي كذلك باستخدام مؤشر إلى البنية:

```go
var employeeOfTheMonth *Employee = &dilbert
employeeOfTheMonth.Position += " (proactive team player)"
```

إن العبارة الأخيرة مكافِئة لـ:

```go
(*employeeOfTheMonth).Position += " (proactive team player)"
```

بعد كتابة الهوية (ID) المتفردة للموظف، تعيد وظيفة EmployeeByID مؤشر لبنية Employee. يمكننا استخدام التدوين النقطي للدخول إلى حقوله:

```
func EmployeeByID(id int) *Employee { /* ... */ }
fmt.Println(EmployeeByID(dilbert.ManagerID).Position) // "Pointy-haired boss"
id := dilbert.ID
EmployeeByID(id).Salary = 0 // fired for... no real reason
```

تحدّث العبارة الأخيرة بنية Employee المشار لها بواسطة نتيجة استدعاء EmployeeByID. لو تغير نوع نتيجة EmployeeByID إلى Employee بدلًا من Employee*، فلن تُترجم عبارة الإسناد لأن جانبها الأيسر لا يحدد متغير.

تُكتب الحقول عادة كحقل في السطر الواحد، ويسبق اسم الحقل نوعه، ولكن الحقول المتتالية لنفس النوع يمكن أن تندمج، كما هو الحال مع الاسم والعنوان في المثال التالي:

```
type Employee struct {
ID              int
Name, Address    string
DoB             time.Time
Position         string
Salary           int
ManagerID        int
}
```

إن ترتيب الحقل مهم بالنسبة لهوية النوع، ولو دمجنا إعلان حقل Position (وهو أيضًا string)، أو بدلنا بين Name و Address، كنا سنضطّر لتعريف نوع بنية مختلفة. نحن ندمج عادة إعلانات الحقول المرتبطة فقط.

يُصدّر اسم حقل البنية لو كان يبدأ بحرف كبير، وهذه هي آلية التحكم الرئيسية في الدخول في Go. قد يحتوي نوع البنية على مزيج من الحقول المُصدّرة وغير المُصدّرة.

تكون أنواع البنية مُطنبة (verbose) لأنها تتضمن سطر لكل حقل عادة، وبالرغم من أننا نستطيع كتابة نوع كامل كل مرة عند الحاجة، إلا أن التكرار سيصبح مُتعِبًا مع مرور الوقت. بدلًا من ذلك، تظهر أنواع البنية عادة داخل إعلان النوع المُسمّى مثل Employee.

إن نوع البنية المُسمّى S لا يمكن أن يعلن عن حقل من نفس النوع :S لأن القيمة المجمعة لا يمكن أن تحتوي على نفسها. (ينطبق قيد مشابه على المصفوفات). لكن S يمكن أن تعلن عن حقل نوعه مؤشر S*، والذي يسمح لنا أن ننشئ بنيات تكرارية، مثل القوائم المرتبطة والأشجار. هذا موضح في الشفرة أدناه، والتي تستخدم شجرة ثنائية لتطبيق فرز للإدخال:

```
gopl.io/ch4/treesort
type tree struct {
    value       int
    left, right *tree
}
// Sort sorts values in place.
```

```
func Sort(values []int) {
   var root *tree
   for _, v := range values {
      root = add(root, v)
   }
   appendValues(values[:0], root)
}
// appendValues appends the elements of t to values in order
// and returns the resulting slice.
func appendValues(values []int, t *tree) []int {
   if t != nil {
      values = appendValues(values, t.left)
      values = append(values, t.value)
      values = appendValues(values, t.right)
   }
   return values
}
func add(t *tree, value int) *tree {
   if t == nil {
      // Equivalent to return &tree{value: value}.
      t = new(tree)
      t.value = value
      return t
   }
   if value < t.value {
      t.left = add(t.left, value)
   } else {
      t.right = add(t.right, value)
   }
   return t
}
```

تتألف القيمة الصفرية للبنية من القيم الصفرية لحقولها. يُفضل عادة أن تكون القيمة الصفرية طبيعة أو قيمة اعتيادية منطقية. كمثال، في bytes.Buffer، القيمة المبدئية للبنية هي صوان فارغ جاهز للاستخدام، والقيمة الصفرية لـ sync.Mutex، والتي سنراها في الفصل التاسع، هي mutex مفتوحة جاهز للاستخدام. يحدث هذا السلوك المبدئي المنطقي هذا بشكل تلقائي أحيانًا، ولكن أحيانًا يجب على مصمم النوع أن يعمل على تحقيقه.

يُطلق على نوع البنية التي لا تحتوي على حقول "بنية فارغة" (empty struct) وتُكتَب {}struct. إن حجمها صفر ولا تحمل أي معلومات ولكنها قد تكون مفيدة رغم ذلك. يستخدمها بعض مبرمجي Go بدلًا من bool كنوع قيمة الخريطة التي تمثل مجموعة، وللتأكيد على أن المفاتيح مهمة فقط، ولكن توفير المساحة هامشي، وبناء الجملة أكثر تعقيدًا، لذا نتجنبها بشكل عام.

```
seen := make(map[string]struct{}) // set of strings
// ...
if _, ok := seen[s]; !ok {
     seen[s] = struct{}{}
     // ...first time seeing s...
}
```

4.4.1 البنيات الحرفية ـ Struct Literals

يمكن كتابة قيمة البنية باستخدام حرف البنية (struct literal) الذي يحدد قيم حقولها.

```
type Point struct{ X, Y int }
p := Point{1, 2}
```

هناك شكلان من أشكال حروف البنية. الشكل الأول هو الموضح أعلاه، ويتطلب تحديد قيمة "كل" حقل، بالترتيب الصحيح. وهو يضع على الكاتب (والقارئ) عبء تذكر الحقول بالضبط، وهي تجعل الشفرة هشة لو نمت مجموعة الحقول لاحقًا أو أعيد ترتيبها. وفقًا لذلك، يُستخدم هذا الشكل عادة فقط داخل الحزم التي تحدد نوع البنية، أو أنواع البنية الأصغر ذات طريقة ترتيب الحقل الواضحة والمعروفة مثل image.Point{x, y} أو orcolor.RGBA{red, green, .blue, alpha}

يُستخدم الشكل الثاني أكثر عادة، وفيه تبدأ قيمة البنية من خلال وضع قائمة ببعض أو كل أسماء الحقل، والقيم المناظرة لها، كما هو الحال في هذه العبارة الموجودة في برنامج Lissajous في القسم 1.4:

```
anim := gif.GIF{LoopCount: nframes}
```

لو حُذف الحقل في هذا النوع البنيات، فإن قيمته تصبح القيمة الصفرية لنوعه، ويصبح ترتيب الحقول غير مهم لأن الأسماء مذكورة.

لا يمكن المزج بين الشكلين في نفس البنية الحرفية. كما لا يمكنك استخدام الشكل الأول (المعتمد على الترتيب) للحروف للتحايل على قاعدة أن المُحدّدات غير المُصدّرة يمكن الإحالة إليها من حزمة أخرى.

```
package p
type T struct{ a, b int } // a and b are not exported
package q
import "p"
var _ = p.T{a: 1, b: 2} // compile error: can't reference a, b
var _ = p.T{1, 2}    // compile error: can't reference a, b
```

بالرغم من أن السطر الأخير أعلاه لا يذكر محددات الحقل غير المُصدّر، إلا أنه يستخدمهم ضمنيًا، لذا يُعَد غير مسموح به.

يمكن تمرير قيم البنية كمعطيات للوظائف، وتُعاد منها. كمثال، تُدرِّج تلك الوظيفة Point بواسطة عامل محدد:

```
func Scale(p Point, factor int) Point {
    return Point{p.X * factor, p.Y * factor}
}
fmt.Println(Scale(Point{1, 2}, 5)) // "{5 10}"
```

لتحقيق الكفاءة، تُمرر أنواع البنية الأكبر حجمًا عادة إلى الوظائف أو تُعاد منها بشكل غير مباشر باستخدام مؤشر،

```
func Bonus(e *Employee, percent int) int {
    return e.Salary * percent / 100
}
```

هذا مطلوب لو كانت الوظيفة يجب أن تعدّل معطياتها، حيث أن اللغات التي تستدعي وفقًا للقيمة مثل Go، تتلقى فيها الوظيفة المُستدعاة نسخة من المعطى فقط وليس مرجع إلى المعطى الأصلي.

```
func AwardAnnualRaise(e *Employee) {
    e.Salary = e.Salary * 105 / 100
}
```

نظرًا لكون البِنيات يتم التعامل معها عبر المؤشرات عادة، يمكن استخدام هذا التدوين المختصر لإنشاء وبدء متغير بنية والحصول على عنوانه:

```
pp := &Point{1, 2}
```

وهذا مطابق تماما إلى:

```
pp := new(Point)
*pp = Point{1, 2}
```

ولكن &Point{1, 2} يمكن استخدامها مباشرة داخل التعبير، مثل استدعاء الوظيفة.

4.4.2 مقارنة البنيات ـ Comparing Structs

لو كانت كل حقول البنية قابلة للمقارنة، فإن البنية نفسها تكون قابلة للمقارنة، وبالتالي يمكن مقارنة تعبيرين من هذا النوع باستخدام == أو != . تقارن عملية == الحقول المتناظرة لبنيتين مرتبتين، وبالتالي سيكون التعبيرين المطبوعين أدناه متكافئين:

```
type Point struct{ X, Y int }

p := Point{1, 2}
q := Point{2, 1}
fmt.Println(p.X == q.X && p.Y == q.Y) // "false"
fmt.Println(p == q)                   // "false"
```

يمكن استخدام أنواع البنية القابلة للمقارنة، مثلها مثل الأنواع القابلة للمقارنة الأخرى، كنوع لمفتاح خريطة.

```
type address struct {
    hostname   string
    port        int
```

```
}
hits := make(map[address]int)
hits[address{"golang.org'', 443}]++
```

4.4.3 تضمين البنية والحقول المجهولة ـ Struct Embedding and Anonymous Fields

سنرى في هذا القسم كيف سمحت لنا آلية "تضمين البنية" (struct embedding) غير المعتادة في Go أن نستخدم نوع بنية مسماة كـ "حقل مجهول" (anonymous field) لنوع بنية أخرى، مما يقدم اختصار تركيبي مريح أكثر، بحيث يمكن لتعبير نقطي بسيط مثل x.f أن يرمز لسلسلة حقول مثل x.d .e.f.

فكر في برنامج رسم ثنائي الأبعاد يقدم مكتبة أشكال، مثل المستطيلات، والأشكال البيضاوية والنجوم والعجلات. إليك اثنين من الأنواع التي يُمكّن أن يعرّفها:

```
type Circle struct {
    X, Y, Radius int
}
type Wheel struct {
    X, Y, Radius, Spokes int
}
```

تحتوي Circle على حقول لإحداثيات X و Y من مركزها، ونصف قُطر، بينما تحتوي Wheel على كل خصائص Circle، إضافة إلى Spokes، وعدد الـ spokes النصف قطرية المحفورة. لنصنع عجلة:

```
var w Wheel
w.X = 8
w.Y = 8
w.Radius = 5
w.Spokes = 20
```

مع نمو مجموعة الأشكال، سنلاحظ تشابهات وتكرارات بينها، وبالتالي قد يكون من الأفضل استبعاد الأجزاء المشتركة بينهم:

```
type Point struct {
    X, Y int
}
type Circle struct {
    Center Point
    Radius int
}
type Wheel struct {
    Circle Circle
```

```
    Spokes int
}
```

قد يكون التطبيق أوضح هنا، ولكن هذا التغيير يجعل الوصول لحقول Wheel مُطنبا أكثر:

```
var w Wheel
w.Circle.Center.X = 8
w.Circle.Center.Y = 8
w.Circle.Radius = 5
w.Spokes = 20
```

تسمح لنا Go بأن نعلن عن الحقل باستخدام النوع ولكن من دون اسم، ويُطلق على تلك الحقول مصطلح "الحقول المجهولة" (anonymous fields). يجب أن يكون نوع الحقل نوعًا مسمى أو مؤشر لنوع مسمى. يحتوي Circle و Wheel في المثال أدناه على حقل مجهول في كل منهما، ونحن نقول أن Point مُضمّن داخل Circle، وأن Circle مُضمّن داخل Wheel.

```
type Circle struct {
    Point
    Radius int
}
type Wheel struct {
    Circle
    Spokes int
}
```

بفضل التضمين، يمكننا الرجوع إلى الأسماء الموجودة في أوراق الشجرة الضمنية بدون منحها أسماء متداخلة:

```
var w Wheel
w.X =8          // equivalent to w.Circle.Point.X = 8
w.Y =8          // equivalent to w.Circle.Point.Y = 8
w.Radius =5     // equivalent to w.Circle.Radius = 5
w.Spokes = 20
```

ما زالت الأشكال الصريحة الموضحة في التعليقات أعلاه سليمة، وهي توضح أن "الحقل المجهول" ناتج عن خطأ في التسمية غالبًا. إن الحقول Circle و Point لهما أسماء بالفعل – وهي أسماء النوع المُسمى – ولكن تلك الأسماء اختيارية في التعبيرات النقطية. يمكن أن نحذف أي أو كل تلك الحقول المجهولة عند اختيار حقولها الفرعية.

لسوء الحظ، لا يوجد اختصار مناظر لتركيب حروف البنية، بالتالي لن يُترجم أي من هؤلاء:

```
w = Wheel{8, 8, 5, 20}     // compile error: unknown fields
w = Wheel{X: 8, Y: 8, Radius: 5, Spokes: 20} // compile error: unknown fields
```

يجب أن يتبع حرف البنية شكل إعلان النوع، وبالتالي يجب أن نستخدم أحد الشكلين أدناه، وهما مكافئين لبعضهما:

```go
gopl.io/ch4/embed
w = Wheel{Circle{Point{8, 8}, 5}, 20}
w = Wheel{
   Circle: Circle{
      Point:  Point{X: 8, Y: 8},
      Radius: 5,
   },
   Spokes: 20, // NOTE: trailing comma necessary here (and at Radius)
}
fmt.Printf("%#v\n", w)
// Output:
// Wheel{Circle:Circle{Point:Point{X:8, Y:8}, Radius:5}, Spokes:20}
w.X = 42
fmt.Printf("%#v\n", w)
// Output:
// Wheel{Circle:Circle{Point:Point{X:42, Y:8}, Radius:5}, Spokes:20}
```

لاحظ كيف يجعل الظرف # الفعل v% في Printf يعرض قيما في شكل مشابه لتركيب Go. أما بالنسبة لقيم البنية، فإن هذا الشكل يتضمن اسم كل حقل.

نظرا لكون الحقول "المجهولة" لها أسماء ضمنية، لا يمكن أن يوجد حقلان مجهولان من نفس النوع، وإلا سيحدث تضارب بين أسماءهم. ونظرًا لكون اسم الحقل يتحدد ضمنًا وفقًا لنوعه، سيتحدد ظهور الحقل وفقًا لنوعه كذلك. إن حقول Point و Circle المجهولة في المثال أعلاه مُصدّرة، ولو كانت غير مصدرة (point و circle)، سيظل بإمكاننا استخدام الشكل المختصر:

```go
w.X = 8 // equivalent to w.circle.point.X = 8
```

ولكن الشكل الطويل الصريح في التعليق سيكون ممنوعًا خارج إعلان الحزمة لأنه سيكون من غير الممكن الدخول إلى circle و point.

إن ما رأيناه حتى الآن في تضمين البنية هو مجرد بداية ونبذة تركيبية مختصرة حول التدوين النقطي المستخدَم لاختيار حقول البنية. سنرى لاحقًا أن الحقول المجهولة لا يجب أن تكون أنواع بنية، فأي نوع مسمى أو مؤشر يشير لنوع مسمى سيفي بالغرض، ولكن لماذا قد ترغب في تضمين نوع ليس به حقول فرعية؟

إن الإجابة متعلقة بالطُرق (Methods)، لأن التدوين المختصر المُستخدَم في اختيار حقول النوع المُضمّن يُستخدَم في اختيار الطُرق أيضًا. في الواقع، يكتسب نوع البنية الخارجية طُرق النوع المُضمّن أيضًا وليس حقوله فقط. إن هذه الآلية هي الطريقة الرئيسية التي تتركب بها سلوكيات الكائن المعقدة من سلوكيات أبسط. يُعد التركيب (Composition) محوريًا للبرمجة كائنية التوجه في Go، وسنستكشف هذا أكثر في القسم 6.3.

4.5 JSON

إن تدوين كائن جافاسكريبت (JSON) هو تدوين قياسي لإرسال واستقبال المعلومات المبنية. ليس JSON التدوين الوحيد من هذا النوع، فـ XMK (انظر 7.14)، و ASN.1 وصوانات بروتوكول جوجل، تخدم كلها أهداف مشابهة، وكل منها له مجاله الخاص، ولكن JSON هو الأكثر استخدامًا بسبب بساطته وسهولته ودعمه العالمي.

تمتلك Go دعما ممتاز لترميز وفك ترميز هذه الصيغة، ويُقدّم هذا الدعم من حزم المكتبة القياسية encoding/json و encoding/xml و encoding/asn1 إلخ، وهذه الحزم كلها تحتوي على APIs متشابهة. يقدم هذا القسم نظرة عامة موجزة على أهم أجزاء حزمة encoding/json.

تُرمّز JSON قيم جافاسكريبت – السلاسل والأرقام والقيم المنطقية ، والمصفوفات، والكائنات – كنص Unicode. إنها تمثيل كفء وسهل القراءة لأنواع البيانات الأساسية المذكورة في الفصل الثالث، والأنواع المُركبة الموجودة في هذا الفصل – المصفوفات والشرائح والبنيات والخرائط.

إن أنواع JSON الأساسية هي أرقام (سواء بتدوين عشري أو علمي)، وقيم منطقية (صحيحة أو خاطئة)، وسلاسل، وهي كلها تسلسلات لنقاط شفرة Unicode المغلفة بين علامتيّ اقتباس مزدوجتين، حيث الخط المائل العكسي يخلُص باستخدام تدوين مشابه في Go، بالرغم من أن خلوص Uhhhh\ الرقمي في JSOM يرمز إلى شفرات UTF-16 وليس إلى runes.

يمكن دمج هذه الأنواع الأساسية بطريقة تكرارية باستخدام مصفوفات وكائنات JSON. إن مصفوفة JSON هي تسلسل منظم للقيم، مكتوب في قائمة يفصل بين كلامها فاصلة، ومغلفة بين أقواس مربعة. تُستخدم مصفوفات JSON لترميز مصفوفات وشرائح Go. إن كائن JSON هو خريطة من السلاسل إلى القيم، مكتوبة كتسلسل لثنائيات name:value يفصل بينها فاصلات، ومحاطة بأقواس. تُستخدم كائنات JSON لترميز خرائط Go (مع مفاتيح سلسلة) وبنيات.

كمثال:

```
boolean         true
number          -273.15
string          "She said \"Hello, 世界\""
array           ["gold", "silver", "bronze"]
object          {"year":1980,
                 "event": "archery",
                 "medals": ["gold", "silver", "bronze"]}
```

فكر في تطبيق يجمع مراجعات الأفلام ويقدم توصيات. إنه من نوع البيانات Movie، وقائمة القيم التقليدية المعلنة الخاصة به موضحة أدناه. (حروف السلسلة بعد إعلانات حقل Year و Color هي "وسوم حقل" (field tags)، وسنشرحها خلال لحظات).

```
gopl.io/ch4/movie
type Movie struct {
   Title  string
   Year   int  `json:"released"`
   Color  bool `json:"color,omitempty"`
   Actors []string
}
var movies = []Movie{
   {Title: "Casablanca", Year: 1942, Color: false,
      Actors: []string{"Humphrey Bogart", "Ingrid Bergman"}},
   {Title: "Cool Hand Luke", Year: 1967, Color: true,
      Actors: []string{"Paul Newman"}},
   {Title: "Bullitt", Year: 1968, Color: true,
      Actors: []string{"Steve McQueen", "Jacqueline Bisset"}},
   // ...
}
```

إن بنيات البيانات المشابهة لهذه ملائمة بشكل ممتازة لـ JSO، ومن السهل تحويلها في كلا الاتجاهين. إن تحويل بنية بيانات Go مثل الأفلام إلى JSON يُطلق عليه "التنظيم" أو Marshaling. تتم عملية الـ Marshaling باستخدام json.Marshal:

```
data, err := json.Marshal(movies)
if err != nil {
     log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)
```

ينتج عن Marshal شريحة بايت تحتوي على سلسلة طويلة جدًا دون مساحة بيضاء خارجية، وقد ضممنا السطور على بعضها حتى تظهر بشكل مناسب:

```
 [{"Title":"Casablanca","released":1942,"Actors":["Humphrey Bogart","Ingr id
Bergman"]},{"Title":"Cool Hand Luke","released":1967,"color":true,"Ac tors":
["Paul Newman"]},{"Title":"Bullitt","released":1968,"color":true, " Actors":
["Steve McQueen","Jacqueline Bisset"]}]
```

يحتوي هذا التمثيل المضغوط على كل المعلومات، ولكنه صعب القراءة. يقدم تنويع اسمه json.MarshalIndent ناتج مُزاح مُنظم سهل القراءة بالنسبة للبشر. يوجد معطيين إضافيين يُعرّفان سابقة كل سطر في الناتج، وسلسلة لكل مستوى إزاحة:

```
data, err := json.MarshalIndent(movies, "", "    ")
```

```
if err != nil {
	log.Fatalf("JSON marshaling failed: %s", err)
}
fmt.Printf("%s\n", data)
```

إن الشفرة أعلاه تطبع التالي:

```
[
	{
		"Title": "Casablanca",
		"released": 1942,
		"Actors": [
			"Humphrey Bogart",
			"Ingrid Bergman"
		]
	},
	{
		"Title": "Cool Hand Luke",
		"released": 1967,
		"color": true,
		"Actors": [
			"Paul Newman"
		]
	},
	{
		"Title": "Bullitt",
		"released": 1968,
		"color": true,
		"Actors": [
			"Steve McQueen",
			"Jacqueline Bisset"
		]
	}
]
```

تستخدم عملية الـ Marshaling أسماء حقل بنية Go كأسماء حقل لكائنات JSON (من خلال reflection، كما سنرى في القسم 12.6). تُنظم الحقول المُصدّرة فقط، ولهذا السبب نختار الأسماء ذات الحروف الكبيرة لكل أسماء حقول Go.

ربما تكون قد لاحظت أن اسم الحقل Year تغير إلى released في الناتج، وأن Color تغير إلى color. هذا بسبب وسوم الحقل (field tags). إن وسم الحقل هو سلسلة من البيانات الوصفية التي ارتبطت في وقت الترجمة مع حقل بنية:

```
Year int 'json:"released"'
Color bool 'json:"color,omitempty"'
```

إن وسم الحقل قد يكون أي سلسلة حرفية، ولكنه يُفسّر عادة بأنه قائمة من key:"value" بينها مسافة، حيث أنه يحتوي على علامات اقتباس مزدوجة، وتُكتب وسوم الحقل عادة بحروف سلسلة خام. يتحكم مفتاح json في سلوك حزمة encoding/json، وتتبع حزم .../encoding الأخرى هذا التقليد. يحدد الجزء الأول من وسم حقل json اسم JSON بديل لحقل Go. تُستخدم وسوم الحقل عادة لتحديد اسم JSON الاصطلاحي، مثل total_count الذي يُستخدم مع حقل Go اسمه TotalCount. يحتوي وسم Color على خيار إضافي هو omitempty، والذي يشير إلى أنه لن يتم إنتاج أي ناتج لـ JSON لو كان الحقل يحتوي على قيمة صفرية لنوعه (قيمة خاطئة في هذه الحالة)، أو لو كان فارغًا. تأكد من أن ناتج JSON لـ Casablanca، وهو فيلم أبيض وأسود، لا يحتوي على حقل color.

إن عملية الـ marshaling العكسية، التي تفك ترميز JSON وتملأ بنية بيانات Go، يُطلق عليها unmarshaling، وتتم بواسطة json.Unmarshal. إن الشفرة أدناه تقوم بعمل عملية unmarshals لبيانات فيلم JSON وتدخلها في شريحة بنيات حقلها الوحيد هو Title. إن تعريف بنيات بيانات Go بهذه الطريقة يُمكّننا من اختيار أجزاء مُدخلات JSON التي سيفك ترميزها، والأجزاء التي ستحذف. عندما تعيد unmarshal نتائجها، سنجد أنها ملئت الشريحة بمعلومات Title، بينما تجاهلت الأسماء الأخرى في JSON.

```
var titles []struct{ Title string }
if err := json.Unmarshal(data, &titles); err != nil {
     log.Fatalf("JSON unmarshaling failed: %s", err)
}
fmt.Println(titles) // "[{Casablanca} {Cool Hand Luke} {Bullitt}]"
```

تقدم العديد من خدمات الويب واجهة JSON – حيث تقدم طلب لـ HTTP وسيعيد لك المعلومات المطلوبة في صيغة JSON. للتوضيح، لنستعلم عن متتبع إصدار GitHub باستخدام واجهة خدمة الويب الخاصة به. أولا، سنعرف الأنواع والثوابت الضرورية:

```
gopl.io/ch4/github
// Package github provides a Go API for the GitHub issue tracker.
// See https://developer.github.com/v3/search/#search-issues.
package github
import "time"
const IssuesURL = "https://api.github.com/search/issues"
type IssuesSearchResult struct {
    TotalCount int `json:"total_count"`
    Items      []*Issue
}
type Issue struct {
    Number    int
    HTMLURL   string `json:"html_url"`
    Title     string
    State     string
    User      *User
    CreatedAt time.Time `json:"created_at"`
```

```
    Body      string    // in Markdown format
}
type User struct {
    Login   string
    HTMLURL string `json:"html_url"`
}
```

كما ذكرنا من قبل، يجب أن تكون أسماء كل حقول البنية بادئة بحرف كبير، حتى لو لم تكن أسماء JSON الخاصة بها تبدأ بها. مع ذلك، عملية التوفيق التي تربط أسماء JSON مع أسماء بنية Go أثناء unmarshaling تتغير حسب كل حالة، وبالتالي من الضروري فقط استخدام وسم الحقل عندما يكون هناك تسطير سفلي في اسم JSON ولكن ليس في اسم Go. سنختار بحرص الحقول التي سنفك ترميزها، حيث تحتوي نتائج بحث GitHub على معلومات أكثر بكثير مما نستطيع تقديمه هنا.

تقدم وظيفة SearchIssues طلب HTTP، وتفك ترميز النتيجة وتقدمها في شكل JSON. نظرًا لكون مصطلحات الاستعلام التي قدمها المستخدم قد تحتوي على حروف مثل ? و & التي لها معنى خاص في URL، سنستخدم url.QueryEscape لضمان أنها تؤخذ بشكل حرفي.

```
gopl.io/ch4/github
package github
import (
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "strings"
)
// SearchIssues queries the GitHub issue tracker.
func SearchIssues(terms []string) (*IssuesSearchResult, error) {
    q := url.QueryEscape(strings.Join(terms, " "))
    resp, err := http.Get(IssuesURL + "?q=" + q)
    if err != nil {
        return nil, err
    }
    //!-
    // For long-term stability, instead of http.Get, use the
    // variant below which adds an HTTP request header indicating
    // that only version 3 of the GitHub API is acceptable.
    //
    //   req, err := http.NewRequest("GET", IssuesURL+"?q="+q, nil)
    //   if err != nil {
    //       return nil, err
    //   }
    //   req.Header.Set(
    //       "Accept", "application/vnd.github.v3.text-match+json")
    //   resp, err := http.DefaultClient.Do(req)
    //!+
    // We must close resp.Body on all execution paths.
    // (Chapter 5 presents 'defer', which makes this simpler.)
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
```

```
        return nil, fmt.Errorf("search query failed: %s", resp.Status)
    }
    var result IssuesSearchResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        resp.Body.Close()
        return nil, err
    }
    resp.Body.Close()
    return &result, nil
}
```

استخدمت الأمثلة السابقة json.Unmarshal لفك ترميز محتويات شريحة بايت كاملة باعتبارها كيان JSON منفرد. سنضيف المزيد من التنويع من خلال جعل هذا المثال يستخدم أداة فك الترميز بالتدفق (streaming) التي يُطلق عليها json.Decoder، والتي تسمح بفك ترميز العديد من كيانات JSON الآتية من نفس التدفق بطريقة متتابعة، بالرغم من أننا لا نحتاج لتلك الخاصية هنا. كما يمكن أن تتوقع، فإن هناك أداة ترميز تدفق مناظرة اسمها json.Encoder.

يملأ استدعاء Decode نتيجة المتغير، وهناك طرق متنوعة يمكننا استخدامها لتهيئة قيمة النتيجة بشكل بسيط. أبسط طريقة هي تلك الموضحة في أمر الإصدارات أو Issues أدناه، وهي جدول نصي بأعمدة ذات عرض ثابت، ولكننا سنرى في القسم التالي طريقة أكثر دقة تعتمد على القوالب.

```
gopl.io/ch4/issues
// Issues prints a table of GitHub issues matching the search terms.
package main
import (
    "fmt"
    "log"
    "os"
    "gopl.io/ch4/github"
)
//!+
func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("%d issues:\n", result.TotalCount)
    for _, item := range result.Items {
        fmt.Printf("#%-5d %9.9s %.55s\n",
            item.Number, item.User.Login, item.Title)
    }
}
```

تحدد معطيات سطر الأوامر مصطلحات البحث. يستعلم الأمر أدناه عن متتبع إصدار مشروع Go الخاص بقائمة الأخطاء المفتوحة المرتبطة بفك ترميز JSON:

```
$ go build gopl.io/ch4/issues
$ ./issues repo:golang/go is:open json decoder
13 issues:
#5680    eaigner encoding/json: set key converter on en/decoder
#6050  gopherbot encoding/json: provide tokenizer
#8658  gopherbot encoding/json: use bufio
#8462  kortschak encoding/json: UnmarshalText confuses json.Unmarshal
#5901        rsc encoding/json: allow override type marshaling
#9812  klauspost encoding/json: string tag not symmetric
#7872  extempora encoding/json: Encoder internally buffers full output
#9650    cespare encoding/json: Decoding gives errPhase when unmarshalin
#6716  gopherbot encoding/json: include field name in unmarshal error me
#6901  lukescott encoding/json, encoding/xml: option to treat unknown fi
#6384    joeshaw encoding/json: encode precise floating point integers u
#6647    btracey x/tools/cmd/godoc: display type kind of each named type
#4237  gjemiller encoding/base64: URLEncoding padding is optional
```

تحتوي واجهة خدمة ويب GitHub الموجودة على https://developer.github.com/v3/ على العديد من الخصائص التي لا توجد مساحة هنا لذكرها كلها.

تمرين 4.10: عدّل برنامج issues لتقدم نتائج في فئات العمر، مثلًا.. عمر أقل من شهر واحد، وأقل من عام، وأكثر من عام.

تمرين 4.11: قم ببناء أداة تساعد المستخدمين على إنشاء وقراءة وتحديث وحذف إصدارات GitHub من سطر الأوامر، وتستخدم محررهم النصي المفضل عند الحاجة لإدخال نص كبير.

تمرين 4.12: تحتوي قصص الويب المصورة ذات لشعبية xkcd على واجهة JSON. كمثال، الطلب المُقدّم إلى https://xkcd.eom/571/info.0.json سينتج عنه وصف تفصيلي للقصة المصورة 571، وهي واحدة من القصص المفضلة جدًا. قم بتحميل كل URL (مرة واحدة) وابني فهرس محلي. اكتب أداة xkcd تطبع – باستخدام هذا الفهرس – الـ URL والتفريغ النصي لكل قصة مصورة مُطابِقة لمصطلحات البحث المذكورة في سطر الأمر.

تمرين 4.13: خدمة ويب Open Movie Database المعتمدة على JSON تسمح لك بالبحث في https://omdbapi.com عن فيلم باسمه وتحميله ملصق الفيلم. اكتب أداة ملصق تقوم بتحميل صورة ملصق الفيلم المذكور في سطر الأمر.

4.6 HTML النص وقوالب

يقوم المثال السابق بأبسط تهيئة ممكنة، و Printf مناسب لها تمامًا، لكن أحيانًا يجب أن تكون التهيئة أكثر تفصيلا ودقة، لذا من المفضل فصل التهيئة عن الشفرة بشكل كامل. يمكن فعل هذا باستخدام حزم text/template و html/template، والتي توفر آلية تبديل قيم المتغير وتحويلها إلى نص أو قالب HTML.

إن القالب هو سلسلة أو ملف يحتوي على جزء أو أكثر مغلفين بين أقواس مزدوجة، {{...}}، يُطلق عليها إجراءات (actions). تُطبق معظم السلسلة حرفيًا، ولكن الإجراءات تحفز سلوكيات أخرى. يحتوي كل إجراء على تعبير بلغة القالب، وهو تدوين بسيط ولكن قوي لطباعة القيم، واختيار حقول البنية، واستدعاء الوظائف والطُرق، والتعبير عن تدفق التحكم مثل عبارات if-else، وحلقات النطاق، وتمثيل قوالب أخرى. سنقدم أدناه سلسلة قالب بسيطة:

```
gopl.io/ch4/issuesreport
const templ = `{{.TotalCount}} issues:
{{range .Items}}----------------------------------------
Number: {{.Number}}
User:   {{.User.Login}}
Title:  {{.Title | printf "%.64s"}}
Age:    {{.CreatedAt | daysAgo}} days
{{end}}`
```

يطبع هذا القالب أولا عدد الإصدارات المطابقة، ثم يطبع العدد، المستخدم، العنوان، العمر بالأيام لكل واحد. يوجد تدوين للقيمة الحالية داخل الإجراء، ويشار له بـ النقطة أو "dot"، ويُكتب كـ ''.''، نقطة. تشير النقطة مبدئيًا إلى مؤشر القالب، وهو github.IssuesSearchResult في هذا المثال. يتوسع إجراء {{.TotalCount}} ليشمل قيمة حقل TotalCount، المطبوع بالطريقة التقليدية. إن إجراءات {{range .Items}} و {{end}} تخلق حلقة، وبالتالي فإن النص بينهما يتوسع عدة مرات، بينما تصبح النقطة مقيدة بالعناصر المتتابعة لـ Items.

يجعل تدوين | داخل الإجراء نتيجة إحدى العمليات مُعطى لعملية أخرى، وهو مناظر لتوارد صدفة Unix. أما في حالة Title، فإن العملية الثانية هي عملية printf، وهي مرادف مدمج لـ fmt.Sprintf في كل القوالب، وبالنسبة لـ Age، فإن العملية الثانية هي وظيفة daysAgo، والتي تحوّل حقل CreatAt إلى الوقت المنقضي، باستخدام time.Since:

```
func daysAgo(t time.Time) int {
	return int(time.Since(t).Hours() / 24)
}
```

لاحظ أن نوع CreatedAt هو time.Time، وليس سلسلة. بنفس الطريقة، قد يتحكم هذا النوع في تهيئة السلسلة الخاصة به (انظر 2.5) من خلال تعريف طُرق معينة، وقد يُعرّف النوع أيضًا طرق للتحكم في سلوك marshaling و unmarshaling الخاصين بـ JSON فيه. إن قيمة time.Time المنظمة بـ JSON-marshal هي سلسلة ذات شكل قياسي.

إن إنتاج خرج باستخدام قالب هو عملية من خطوتين، الأولى أننا يجب أن نحلل القالب إلى تمثيل داخلي مناسب، ثم ننفذه في مُدخلات محددة. يجب إجراء التحليل مرة واحدة. تقوم الشفرة أدناه بإنشاء وتحليل القالب templ الموضح أعلاه. لاحظ تسلسل استدعاءات الطريقة: تقوم template.New بإنشاء وإعادة قالب، بينما Func تضيف daysAgo إلى مجموعة الوظائف التي يمكن الوصول لها داخل هذا القالب، ثم تعيد هذا القالب، وأخيرًا تُستدعى Parse لتحليل النتيجة.

```
report, err := template.New("report").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(templ)
if err != nil {
    log.Fatal(err)
}
```

نظرًا لكون القوالب ثابتة عادة في وقت الترجمة، يشير الفشل في تحليل القالب إلى وجود عيب خطير في البرنامج. إن وظيفة المساعِد template.Must تجعل التعامل مع الأخطاء عملية مريحة أكثر، فهي تقبل قالب وخطأ، وتبحث ما إذا كان الخطأ nil (وتهلع في حالة كان خلاف ذلك)، ثم تعيد القالب. سنعود إلى هذه الفكرة في القسم 5.9.

بعد إنشاء القالب، وتعزيزه بـ daysAgo، وتحليله، والتحقق منه، يمكننا تنفيذه باستخدام github.IssuesSearchResult كمصدر بيانات، و os.Stdout كواجهة:

```
var report = template.Must(template.New("issuelist").
    Funcs(template.FuncMap{"daysAgo": daysAgo}).
    Parse(templ))
func main() {
    result, err := github.SearchIssues(os.Args[1:])
    if err != nil {
        log.Fatal(err)
    }
    if err := report.Execute(os.Stdout, result); err != nil {
        log.Fatal(err)
    }
}
```

يطبع البرامج تقرير نصي بسيط كهذا:

```
$ go build gopl.io/ch4/issuesreport
$ ./issuesreport repo:golang/go is:open json decoder
13 issues:
----------------------------
Number: 5680
User:     eaigner
Title:    encoding/json: set key converter on en/decoder
Age:      750 days
-----------------------------
```

```
Number: 6050
User:    gopherbot
Title:   encoding/json: provide tokenizer
Age:     695 days
----------------------------
….
```

لنعود الآن إلى حزمة html/template. تستخدم الحزمة نفس API ولغة التعبير الذين تستخدمهم text/template، ولكنها تضيف خصائص لخلوص السلاسل التلقائي والمناسب للسياق الذي يظهر داخل HTML وجافاسكريت و CSS و URLs. يمكن لهذه الخصائص المساعدة على تجنب المشكلة الأمنية الدائمة في إنتاج HTML، وهي مشكلة "injection attack"، والتي يقوم فيها نظام خصم بتركيب قيمة سلسلة، مثل عنوان إصدار، بحيث يتضمن شفرة خبيثة تمنحهم سيطرة على الصفحة، لو لم يقم القالب بتخليصها بشكل مناسب.

يطبع القالب أدناه قائمة بالإصدارات مثل جدول HTML. لاحظ الاستيراد المختلف:

```
gopl.io/ch4/issueshtml
import "html/template"
var issueList = template.Must(template.New("issuelist").Parse(`
<h1>{{.TotalCount}} issues</h1>
<table>
<tr style='text-align: left'>
  <th>#</th>
  <th>State</th>
  <th>User</th>
  <th>Title</th>
</tr>
{{range .Items}}
<tr>
  <td><a href='{{.HTMLURL}}'>{{.Number}}</a></td>
  <td>{{.State}}</td>
  <td><a href='{{.User.HTMLURL}}'>{{.User.Login}}</a></td>
  <td><a href='{{.HTMLURL}}'>{{.Title}}</a></td>
</tr>
{{end}}
</table>
`))
```

ينفذ الأمر أدناه القالب الجديد على نتائج استعلام مختلف بدرجة طفيفة:

```
$ go build gopl.io/ch4/issueshtml
$ ./issueshtml repo:golang/go commenter:gopherbot json encoder >issues.html
```

يوضح الشكل 4.4 شكل الجدول في متصفح الويب. تتصل الروابط مع صفحات الويب المناسبة في GitHub.



17 issues

#	State	User	Title
7872	open	extemporalgenome	encoding/json: Encoder internally buffers full output
5683	open	gopherbot	encoding/json: performance slower than expected
6901	open	lukescott	encoding/json, encoding/xml: option to treat unknown fields as an error
4474	closed	gopherbot	encoding/json: json encoder fails for embedded non-struct fields
4747	closed	gopherbot	encoding/json Added tag options to ignore fields of struct for encoder/decoder separately
7767	closed	gopherbot	encoding/json: Encoder adds trailing newlines
4606	closed	gopherbot	JSON Package fails to properly escape strings
8582	closed	matt-duch	encoding/json: inconsistent behavior in *(numeric type) and string tag option
6339	closed	gopherbot	encoding/json: Marshal of nil net.IP fails
7337	closed	gopherbot	encoding/json: make "json" tag user-settable
11508	closed	josharian	cmd/go: trace http viewer: "http: multiple response.WriteHeader calls"
1017	closed	gopherbot	json crash on {} input
8592	closed	gopherbot	encoding/json: No way to avoid HTMLEscape when Marshal()-ing
7846	closed	gopherbot	encoding/json: Slice created using reflect.MakeSlice() treated as interface{}
2761	closed	gopherbot	Marshaler cannot work with omitempty in encoding/json
1133	closed	gopherbot	encoding/asn1: inconsistent APIs
7841	closed	gopherbot	reflect: reflect.unpackEface reflect/value.go:174 unexpected fault address 0x0

الشكل 4.4. جدول HTML لإصدارات مشروع Go المرتبطة بترميز JSON.

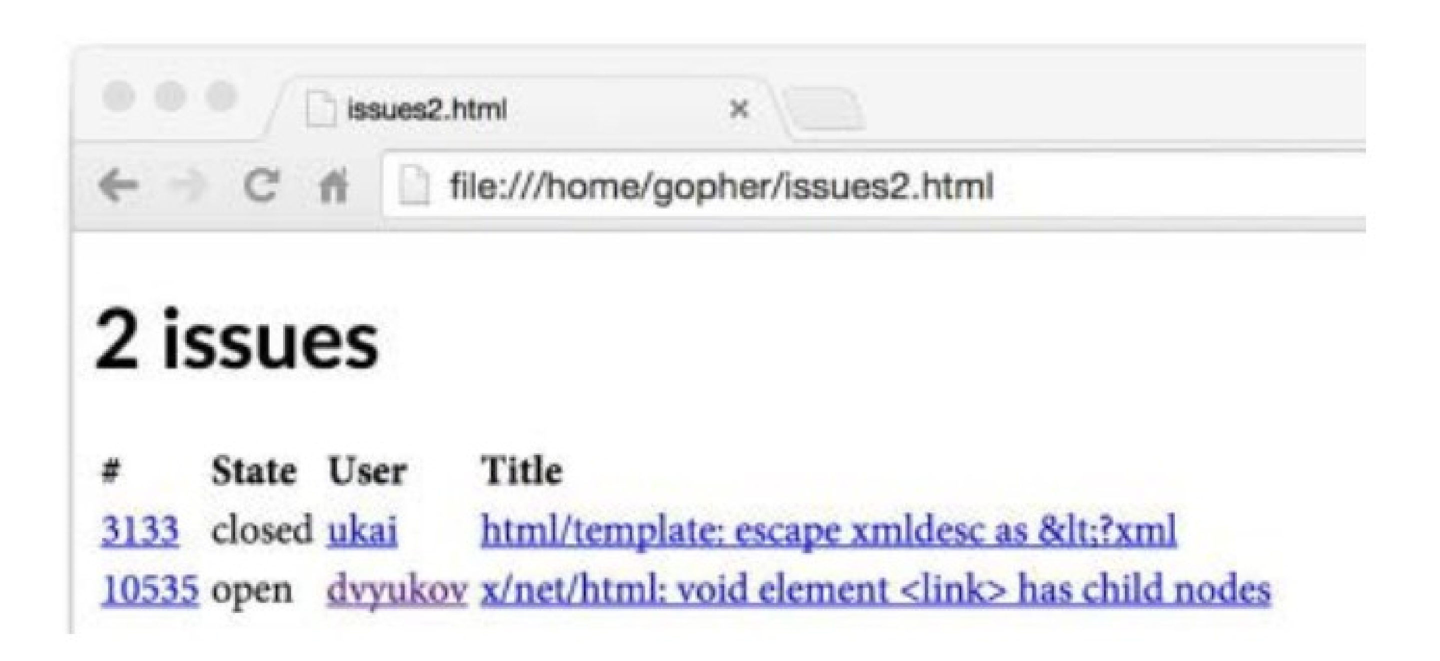
لا يمثل أي من الإصدارات الموجودة في الشكل 4.4 تحديًا لـ HTML، ولكن يمكن أن نرى تأثيرها بوضوح أكبر في الإصدارات التي تحتوي عناوينها على حروف HTML وصفية مثل & و >. لقد اخترنا إصدارين من هذا النوع في المثال:

```
$ ./issueshtml repo:golang/go 3133 10535 >issues2.html
```

يوضح الشكل 4.5 نتيجة هذا الاستعلام. لاحظ أن حزمة html/template خلَّصت العناوين بصيغة HTML تلقائيًا بحيث تظهر بشكل حرفي. لو كنا استخدمنا حزمة text/template عن طريق الخطأ، فإن السلسلة المُكونة من أربع حروف "<" كانت ستُعتبر أقل من حرف واحد '<'، والسلسلة "<link>" كانت ستصبح عنصر رابط، وتغير بنية مستند HTML، وربما تعرض أمن الملف للخطر.

يمكننا كبت سلوك التخليص التلقائي هذا في الحقول التي تحتوي على بيانات HTML موثوق فيها باستخدام نوع سلسلة مسمى template.HTML، بدلًا من سلسلة فقط. توجد أنواع مسماة مشابهة لجافاسكريبت و CSS و URLs الموثوق فيها.

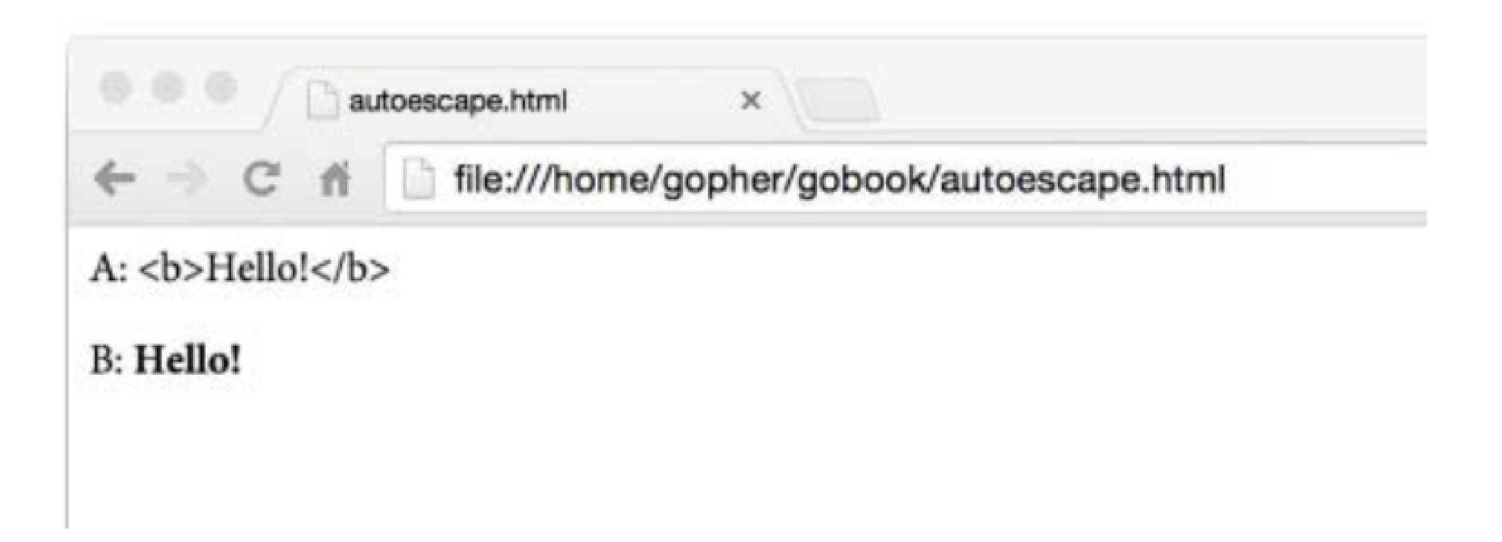
يوضح البرنامج أدناه هذا المبدأ باستخدام حقلين لهما نفس القيمة ولكن بنوعين مختلفين: A هو سلسلة و B هو template.HTML.



الشكل 4.5: حروف HTML الوصفية في عناوين الإصدارات معروضة بشكل صحيح.

```
gopl.io/ch4/autoescape
func main() {
    const templ = `<p>A: {{.A}}</p><p>B: {{.B}}</p>`
    t := template.Must(template.New("escape").Parse(templ))
    var data struct {
        A string        // untrusted plain text
        B template.HTML // trusted HTML
    }
    data.A = "<b>Hello!</b>"
    data.B = "<b>Hello!</b>"
    if err := t.Execute(os.Stdout, data); err != nil {
        log.Fatal(err)
    }
}
```

يوضح الشكل 4.6 ناتج القالب كما يظهر في المتصفح. يمكننا أن نرى أن A تعرض للتخليص، ولكن B لم يتعرض له.



الشكل 4.6: قيمة السلسلة تم تخليصها كـ HTML ولكن قيم template.HTML لم تُخلص.

لدينا مساحة هنا لتوضيح السمات الأساسية فقط لنظام القوالب، وكما هو الحال دائمًا، لو أردت معرفة المزيد من المعلومات، يمكنك الرجوع لتوثيق الحزمة.

```
$ go doc text/template
$ go doc html/template
```

تمرين 4.14: قم بإنشاء خادم ويب يستعلم GitHub مرة واحدة، ثم يسمح بتصفح قائمة تقارير الخطأ، والعلامات الفارقة، والمستخدمين.

5- الوظائف Functions

تُمكِننا الوظائف أو الدوال من تغليف مجموعة من العبارات وجعلها وحدة واحدة يُمكن استدعائها من أي مكان في البرنامج، ويمكن أن تُستدعى أكثر من مرة. وهي تتيح إمكانية تقسيم العمل الكبير إلى أجزاء أصغر قد يكتبها أشخاص مختلفون في أماكن وأوقات مختلفة. تخفي الوظائف أيضًا تفاصيل تطبيقها عن مستخدميها. من أجل هذه الأسباب مجتمعة, تُعَد الوظائف جزءا مهما للغاية في أي لغة برمجة.

لقد رأينا العديد من الوظائف بالفعل. سنأخذ الآن بعض الوقت للحديث عن الوظائف بتفصيل أكبر. المثال الذي ستعمل عليه في هذا الفصل "زاحف ويب" (web crawler)؛ وهو جزء من محركات بحث الويب المسؤول عن جلب صفحات الويب واستكشاف الروابط داخلها، وجلب الصفحات التي تحددها هذه الروابط، إلخ. يمنحنا زاحف الويب فرصة أكثر من كافية لاستكشاف "التكرار" أو (recursion)، واستكشاف الوظائف المجهولة والتعامل مع الأخطاء والجوانب الفريدة في وظائف جو.

5.1 إعلانات الوظائف - Function Declarations

يتكون إعلان الوظيفة من اسم وقائمة من المُعاملات وقائمة اختيارية من النتائج وجسم:

```
func name(parameter-list) (result-list) {
    body
}
```

تحدد قائمة المُعاملات أسماء وأنواع معاملات الوظيفة، وهي عبارة عن متغيرات محلية يقدم المستدعي قيمتها أو معطياتها. تحدد قائمة النتائج أنواع النتائج التي ستعيدها الوظيفة. إذا كانت الوظيفة تُرجع نتيجة واحد غير مُسمى "unnamed result" أو لا تُرجع نتائج على الإطلاق، فستكون الأقواس اختيارية وفي الغالب تُهمل. ترك قائمة النتائج بشكل كامل يؤدي للإعلان عن وظيفة لا تُرجع أي قيمة وتُستدعى فقط من أجل التأثير الذي تُحدثه. كمثال، في وظيفة hypto:

```
func hypot(x, y float64) float64 {
    return math.Sqrt(x*x + y*y)
}
fmt.Println(hypot(3, 4)) // "5"
```

x و y مُعاملات في الإعلان، ٣ و ٤ معطيات في الاستدعاء، والوظيفة تُرجع قيمة من نوع float64 "فاصلة عائمة".

يُمكن أن نسمى نتيجة الوظيفة مثلما نفعل مع المعاملات. في هذه الحالة فإن كل اسم يُعلن عن متغير محلي قيمته المبدئية هي القيمة الصفرية لنوعه.

الوظيفة التي تحتوي على قائمة للنتائج يجب أن تنتهي بعبارة return إلا إذا كان تنفيذ البرنامج لا يستطيع أن يصل إلى نهاية الوظيفة، ربما لأن الوظيفة تنتهي باستدعاء panic أو حلقة for loop لا نهائية بدون عبارة break.

كما رأينا في وظيفة hypot، يُمكن تحليل عوامل تسلسل معاملات أو نتائج من نفس النوع، بحيث يُكتب النوع نفسه مرة واحدة فقط. إن هذين الإعلانين متماثلان:

```
func f(i, j, k int, s, t string)     { /* ... */ }
func f(i int, j int, k int, s string, t string) { /* ... */ }
```

و هذه أربعة طرق للإعلان عن وظيفة ذات معاملين ونتيجة وحيد، وكلهم من نوع int. يمكن استخدام المُعرّف الفارغ للتأكيد على أن المعامل غير مستخدَم.

```
func add(x int, y int) int      { return x + y   }
func sub(x, y int) (z int)      { z = x - y; return }
func first(x int, _ int) int    { return x }
func zero(int, int) int         { return 0 }
fmt.Printf("%T\n", add)    // "func(int, int) int"
fmt.Printf("%T\n", sub)    // "func(int, int) int"
fmt.Printf("%T\n", first) // "func(int, int) int"
fmt.Printf("%T\n", zero)   // "func(int, int) int"
```

قد تسمى الوظيفة أحيانًا باسم توقيعها، ويكون لوظيفتين نفس النوع أو التوقيع لو كانا يمتلكان نفس تسلسل أنواع المعاملات ونفس تسلسل أنواع النتيجة. لا تؤثر أسماء المعاملات والنتائج على النوع، ولا على ما إذا كانت تُعلن باستخدام شكل محلَّل عامليًا أم لا.

يجب أن يوفر كل استدعاء لوظيفة ما معطى لكل معامل، بالترتيب الذي تُعلن به المعاملات. لا تحتوي Go على مفهوم قيم المعامل الافتراضية، ولا على أي طريقة لتحديد المعطيات بالاسم، وبالتالي فإن أسماء المعاملات والنتائج غير مهمة للمستدعي إلا عند التوثيق فقط.

إن المعامِلات هي متغيرات محلية داخل جسد الوظيفة، قيمها المبدئية محددة بالمعطيات التي يقدمها المستدعي. إضافة إلى هذا، فإن معاملات الوظيفة والنتائج المسماة هم متغيرات توجد في نفس الكتلة اللغوية الخاصة بالمتغيرات المحلية الخارجية للوظيفة.

تمرر المعطيات بالقيمة، وبالتالي تتلقى الوظيفة نسخة من كل مُعطى، ولا يتأثر المستدعي بالتعديلات التي تُجرى على النسخة. مع ذلك، لو كان المُعطى يحتوي على نوع من المراجع، كمؤشر أو شريحة أو خريطة أو وظيفة أو قناة، فقد يتأثر المستدعي بأي تعديلات تُجريها الوظيفة على المتغيرات التي يشير لها المُعطى بشكل غير مباشر.

قد تواجه من حين لآخر إعلانات وظيفة دون جسم، وهذا يشير إلى أن الوظيفة تُطبق بلغة أخرى غير لغة Go. يُعرّف هذا الإعلان توقيع الوظيفة.

```
package math
func Sin(x float64) float64 // implemented in assembly language
```

5.2 التكرار - Recursion

يُمكن أن تكون الوظيفة تكرارية (recursive)، وهذا يعني أن بإمكانها استدعاء نفسها، سواء أكان هذا بشكل مباشر أم غير مباشر. يُعتبر التكرار تقنية قوية في التعامل مع العديد من المشكلات، وهي أساسية بالتأكيد في معالجة بنيات البيانات التكرارية (recursive data structures). استخدمنا في القسم 4.4 التكرار في شجرة بيانات لتطبيق ترتيب إقحام (insertion sort) بسيط. سنستخدمه في هذا القسم مرة أخرى لمعالجة مستندات HTML.

يستخدم البرنامج المُقدّم في المثال أدناه حزمة غير قياسية، golang.org/x/net/html، تُوفر محلل HTML. تحمل مستودعات ...golang.org/x/ حزم صممها وصانها فريق Go المسئول عن تطبيقات مثل الشبكات، ومعالجة النصوص المُدوَّلة (internationalized text processing) ومنصات الهواتف النقالة، والتلاعُب بالصور(image manipulation) والتعمية (cryptography) وأدوات المُطورين. لا توجد هذه الحزم في المكتبة القياسية إما لأنها ما زالت تحت التطوير أو لأن مُطوري Go لا يحتاجونها إلا نادرًا.

موضَّح أدناه أجزاء الواجهة البرمجية (API) لـ golang.org/x/net/html التي سنحتاجها. تقرأ الوظيفة html.Parse سلسلة من البايتات (bytes)، وتحللها، وتعيد جذر شجرة مستند HTML، والذي يوجد في html.Node. تحتوي HTML على العديد من أنواع العُقَد (nodes) – مثل النصوص (texts) والتعليقات (comments) وغيرها - لكننا سنركز هنا فقط على عُقَد العنصر التي على شكل <name key='value'>.

```
golang.org/x/net/html
package html
type Node struct {
    Type                    NodeType
    Data                    string
    Attr                    []Attribute
    FirstChild, NextSibling *Node
}
```

```
type NodeType int32 const (
    ErrorNode NodeType = iota
    TextNode
    DocumentNode
    ElementNode
    CommentNode
    DoctypeNode
type Attribute struct {
     Key, Val string
}
func Parse(r io.Reader) (*Node, error)
```

تحلل الوظيفة main المُدخل القياسي باعتباره HTML، وتستخلص الروابط باستخدام وظيفة visit التكرارية، وتطبع كل الروابط المُكتشفة:

```
gopl.io/ch5/findlinks1
// Findlinks1 prints the links in an HTML document read from standard input.
package main
import (
    "fmt"
    "os"
    "golang.org/x/net/html"
)
func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "findlinks1: %v\n", err)
        os.Exit(1)
    }
    for _, link := range visit(nil, doc) {
        fmt.Println(link)
    }
}
```

تخترق الوظيفة visit شجرة عقدة HTML، وتستخلص الرابط من الخاصية href من كل عنصر مربط (anchor) <a href='...'>، وتُلحق الرابط بشريحة من السلاسل، وتعيد الشريحة الناتجة:

```
// visit appends to links each link found in n and returns the result.
func visit(links []string, n *html.Node) []string {
    if n.Type == html.ElementNode && n.Data == "a" {
        for _, a := range n.Attr {
            if a.Key == "href" {
                links = append(links, a.Val)
            }
        }
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        links = visit(links, c)
    }
    return links
}
```

تستدعي الوظيفة visit نفسها تكرارًا لكل طفل من أطفال n، الموجودين في قائمة FirstChild ذات الروابط، كي تتمكن من نزول الشجرة حتى الوصول للعقدة n.

دعنا نشغل الوظيفة findlinks على الصفحة الرئيسية للغة Go، ونُمرر مُخرجات fetch (انظر 1.5) إلى مُدخلات findlinks. لقد عدلنا المُخرجات قليلًا لاختصارها.

```
$ go build gopl.io/chl/fetch
$ go build gopl.io/ch5/findlinks1
$ ./fetch https://golang.org | ./findlinksl #
/doc/
/pkg/
/help/
/blog/
http://play.golang.org/
//tour.golang.org/
https://golang.org/dl/
//blog.golang.org/
/LICENSE /doc/tos.html
http://www.google.com/intl/en/policies/privacy/
```

لاحظ تنوع أشكال الروابط التي تظهر في الصفحة. سنرى لاحقًا كيف يمكننا تحليلهم بالنسبة لـ URL الأساسي: ,URL https ://golang.org، لعمل URLs مُطلقة.

يستخدم البرنامج التالي التكرار في شجرة عقدة HTML لطباعة مخطط الشجرة (tree outline)، وعندما يواجه البرنامج كل عنصر، يدفع وسم العنصر إلى رصة (stack)، ثم يطبع الرصة.

```
gopl.io/ch5/outline
func main() {
    doc, err := html.Parse(os.Stdin)
    if err != nil {
        fmt.Fprintf(os.Stderr, "outline: %v\n", err)
        os.Exit(1)
    }
    outline(nil, doc)
}
func outline(stack []string, n *html.Node) {
    if n.Type == html.ElementNode {
        stack = append(stack, n.Data) // push tag
        fmt.Println(stack)
    }
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        outline(stack, c)
    }
}
```

لاحظ أحد الأشياء الدقيقة: بالرغم من أن المخطط "يضغط" العنصر في رصة، إلا أنه لا يوجد عنصر منبثق مناظر له. عندما يستدعي المخطط نفسه تكراريًا، يتلقى المستدعَى نسخة من الرصة، وبالرغم من أن المستدعَى قد يُلحق عناصر بهذه الشريحة، ويعدل مصفوفتها الضمنية، وربما يخصص مصفوفة جديدة تمامًا حتى، إلا أنه لا يُعدّل العناصر المبدئية الظاهرة للمستدعِي، لذا عندما تعود الوظيفة، تكون رصة المستدعي كما كانت قبل الاستدعاء.

هذا مخطط عام لـ https://golang.org وقد عدلناه لاختصاره مرة أخرى:

```
$ go build gopl.io/ch5/outline
$ ./fetch https://golang.org | ./outline
[html]
[html head]
[html head meta]
[html head title]
[html head link]
[html body]
[html body div]
[html body div]
[html body div div]
[html body div div form]
[html body div div form div]
[html body div div form div a]
...
```

كما ترى من تجربة الوظيفة outline، أغلب مستندات HTML يُمكن معالجتها بمستويات قليلة من التكرار، لكن ليس من الصعب إنشاء مسارات صفحات وِيب تحتاج تكرار أكثر عمقًا بشكل كبير.

تستخدم العديد من تطبيقات لغات البرمجة رصة استدعاء وظيفة ذات حجم ثابت، وتتراوح الأحجام التقليدية من 64 كيلو بايت إلى 2 ميجا بايت. تضع الرصات ثابتة الحجم حدًا على عمق التكرار، لذا يجب علينا أن نتجنب الرص الفائض (stack overflow) عند اختراق بنيات بيانات كبيرة بشكل تكراري؛ لأن الرصة ثابتة الحجم يُمكن أن تُسبب مشكلة أمنية. على النقيض، تستخدم تطبيقات Go التقليدية رصّات متباينة الحجم، تبدأ صغيرة وتكبر حسب الحاجة بحد أقصى جيجا بايت. يجعلنا هذا نستخدم التكرار بأمان دون القلق من الفيض (overflow).

تمرين 5.1: غيّر برنامج findlinks بحيث يخترق قائمة n.FirstChild ذات الروابط باستخدام استدعاءات تكرارية لوظيفة visit بدلًا من استخدام حلقة (loop).

تمرين 5.2: اكتب وظيفة لملء خريطة من أسماء العناصر – p و div و span إلخ – إلى عدد العناصر التي تحمل هذا الاسم في شجرة مستند HTML.

تمرين 5.3: اكتب وظيفة تطبع محتويات كل عُقد النص في شجرة مستند HTML. لا تنحدر إلى عناصر <script> أو <style>، حيث أن محتوياتهم لا تظهر في متصفح الويب.

تمرين 5.4: وسّع وظيفة visit بحيث تستخلص أنواع روابط أخرى من المستند، مثل الصور وسكريبتات وصفحات الأساليب (style sheets).

5.3 إعادة قيم متعددة - Multiple Return Values

يمكن أن تعيد الوظيفة أكثر من نتيجة واحدة. لقد رأينا العديد من الأمثلة لوظائف من حزم قياسية تقوم بإعادة قيمتين، القيمة الحسابية المرغوبة و قيمة خطأ أو قيمة منطقية، والتي تُحدد ما إذا كانت العملية الحسابية نجحت أم لا. يوضح لنا المثال التالي كيف نكتب واحدة خاصة بنا.

إن البرنامج أدناه هو تنويع على findlinks وهو يجعل HTML يطلب نفسه بحيث لا نعود بحاجة إلى إجراء fetch، ونظرًا لكون عمليات HTTP والتحليل يمكن أن تفشل، يعلن findLinks عن نتيجتين: الأولى هي قائمة بالروابط المكتشفة، والثانية نتيجة الخطأ. قد يتعافى محلل HTML عادة من المُدخل السيء ويبني مستند يحتوي على عُقد الخطأ، وبالتالي نادرًا ما تفشل Parse، وعندما تفشل، يرجع هذا غالبًا إلى أخطاء I/O الضمنية.

```
gopl.io/ch5/findlinks2
func main() {
    for _, url := range os.Args[1:] {
        links, err := findLinks(url)
        if err != nil {
            fmt.Fprintf(os.Stderr, "findlinks2: %v\n", err)
            continue
        }
        for _, link := range links {
            fmt.Println(link)
        }
    }
}
// findLinks performs an HTTP GET request for url, parses the
// response as HTML, and extracts and returns the links.
func findLinks(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }
```

```
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    return visit(nil, doc), nil
}
```

يوجد أربع عبارات إعادة في findLinks، كل منها تعيد زوجا من القيم. إن أول ثلاث إعادات تجعل الوظيفة تمرر الأخطاء الضمنية القادمة من حزم http و html إلى المستدعي. في الحالة الأولى، يُعاد الخطأ دون تغيير، بينما في الحالتين الثانية والثالثة، يُعزز الخطأ بمعلومات سياق إضافية تقدمها fmt.Errorf (انظر 7.8). لو كان findLinks ناجحًا، فإن عبارة الإعادة الأخيرة تعيد شريحة روابط بدون أخطاء.

يجب أن نتأكد أن resp.Body أغلقت حتى نضمن أن موارد الشبكة ستحرر بشكل سليم حتى عند حدوث خطأ ما. يقوم جامع نفايات Go بإعادة تدوير الذاكرة غير المستخدمة، ولكن لا تفترض أنه سيحرر موارد النظام التشغيلية غير المستخدَمة مثل الملفات المفتوحة واتصالات الشبكة، بل يجب إغلاق هؤلاء بشكل صريح.

إن نتيجة استدعاء وظيفة متعددة القيمة هي صف من القيم، ويجب على مستدعي هذه الوظيفة تخصيص القيم لمتغيرات بشكل صريح لو أراد استخدام أي منهم.

```
links, err := findLinks(url)
```

لو أردت تجاهل إحدى القيم، قم بتخصيصها لمُعرّف فارغ:

```
links, _ := findLinks(url) // errors ignored
```

إن نتيجة الاستدعاء متعدد القيم نفسه يمكن أن تعود من وظيفة استدعاء (متعددة القيم) أيضًا، كما هو الحال في الوظيفة التالية التي تعمل مثل findLinks ولكنها تسجل معطياتها:

```
func findLinksLog(url string) ([]string, error) {
     log.Printf("findLinks %s", url)
     return findLinks(url)
}
```

قد يظهر الاستدعاء متعدد القيم كمُعطى أحادي عند استدعاء وظيفة متعدد المعامِلات، وبالرغم من أن هذه الخاصية نادرًا ما تُستخدم في شفرة الإنتاج،إلا أنها أحيانًا ما تكون مهمة أثناء التنقيح (debugging) لأنها تُمكّنا من طباعة كل نتائج الاستدعاء باستخدام عبارة واحدة. إن جملتي print التاليتين لهما نفس التأثير.

```
log.Println(findLinks(url))

links, err := findLinks(url)
```

```
log.Println(links, err)
```

يمكن للأسماء المختارة جيدًا أن توثق أهمية نتائج الوظيفة، كما تُعَد الأسماء قيّمة بشكل خاص عندما تعيد الوظيفة نتائج متعددة من نفس النوع، مثل:

```
func Size(rect image.Rectangle) (width, height int)
func Split(path string) (dir, file string)
func HourMinSec(t time.Time) (hour, minute, second int)
```

لكن ليس من الضروري دائمًا تسمية نتائج متعددة بهدف التوثيق فقط. كمثال، ينص العرف السائد على أن النتيجة المنطقية النهائية تشير للنجاح، وأن نتيجة الخطأ لا تحتاج لشرح أو تفسير.

و لكن ليس دائما علينا أن نسمي فقط من أجل التوثيق. على سبيل المثال، جرى العرف أن الناتج الأخير ناتج منطقي bool يدل على النجاح، و الخطأ error ليس بحاجة لتوصيف.

يمكن حذف معاملات عبارة الإعادة في الوظائف ذات النتائج المسماة. هذا يُسمى بالإعادة الخالية bare return.

```
// CountWordsAndImages does an HTTP GET request for the HTML
// document url and returns the number of words and images in it.
func CountWordsAndImages(url string) (words, images int, err error) {
    resp, err := http.Get(url)
    if err != nil {
        return
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        err = fmt.Errorf("parsing HTML: %s", err)
        return
    }
    words, images = countWordsAndImages(doc)
    return
}
func countWordsAndImages(n *html.Node) (words, images int) { /* ... */ }
```

إن الإعادة الخالية هي أسلوب مختصر لإعادة كل متغير من متغيرات النتيجة المسماة بالترتيب. من ثم، فإن كل عبارة إعادة في الوظيفة أعلاه، تكافئ التالي:

```
return words, images, err
```

في الوظائف المشابهة لهذه الوظيفة، التي تحتوي على العديد من عبارات الإعادة والعديد من النتائج، يمكن للإعادات الخالية أن تقلل تكرار الشفرة، ولكنها نادرًا ما تسهّل فهم الشفرة. كمثال، ليس من الواضح من النظرة الأولى أن الإعادتين المبكرتين مكافئتين لإعادة 0, 0, err (لأن متغيرات النتيجة words و images يبدآن بقيمهما الصفرية)، وأن الإعادة النهائية مكافئة لإعادة: words, images, nil. لهذا السبب، من الأفضل استخدام الإعادات الخالية عند الضرورة فقط.

تمرين 5.5: طبق countWordsAndImages. (انظر تمرين 4.9 الخاص بتقسيم الكلمات).

تمرين 5.6: عدل الوظيفة corner الموجودة في gopl.io/ch3/surface (انظر 3.2) بحيث تستخدم النواتج المسماة والإعادة الخالية.

5.4 الأخطاء - Errors

تنجح بعض الوظائف في مهمتها دائمًا. مثل strings.Contains و strconv.FormatBool التي تقدم نتائج جيدة جدًا لكل قيم المعطيات المختلفة ولا يمكن أن تفشل، وتمنع سيناريوهات كارثية ولا يمكن توقعها مثل نفاذ الذاكرة، حيث العَرض يظهر بعيدًا عن سببه، ولا يوجد أي أمل في التعافي من تلك الكارثة عند وقوعها.

تنجح بعض الوظائف الأخرى دائمًا طالما توفرت شروطها. كمثال، وظيفة time.Date تبني دائمًا time.Time من مكوناتها – السنة والشهر إلخ – ما لم يكن المعطى الأخير (وهو المنطقة الزمنية) قيمته nil، لأن هذا يُحدث حالة هلع. إن هذا الهلع هو علامة أكيدة على وجود خطأ في شفرة الاستدعاء، ولا يجب أن يحدث أبدًا في البرنامج المكتوب جيدًا.

أما بالنسبة للعديد من الوظائف الأخرى، لا يكون نجاح البرامج أكيدًا، حتى البرامج المكتوبة جيدًا، لأن نجاحها يعتمد على عوامل ليست تحت سيطرة المبرمج. إن أي وظيفة تقوم بـ I/O مثلا، يجب أن تواجه احتمالية الخطأ، والمبرمج الساذج فقط هو الذي يصدق أن أمر read أو write بسيط لا يمكن أن يفشل. نحن نحتاج لمعرفة السبب أكثر عندما تفشل العمليات الموثوقة بشكل غير متوقع.

من ثمّ، تُعتبر الأخطاء جزءا مهمًا من API الحزمة أو واجهة مستخدم التطبيق، والفشل فيها هو أحد السلوكيات العديدة المتوقعة. هذا هو المنهج الذي تطبقه Go في التعامل مع الأخطاء.

إن الوظيفة التي يُعَد الفشل سلوكًا متوقعًا فيها تعيد نتيجة إضافية، وعادة ما تكون النتيجة الأخيرة. لو كان الفشل له سبب وحيد محتمل، فإن النتيجة تكون قيمة منطقية (بوليان) يُطلق عليها ok عادة، كما هو مبين في المثال التالي الخاص بالبحث في مخبأ (cache) الذي ينجح دائمًا إلا لو لم يكن هناك مدخَل لهذا المفتاح:

```
value, ok := cache.Lookup(key) if !ok {
    // ...cache[key] does not exist...
}
```

يرجع الفشل لأسباب متعددة عادةً خاصة عندما يتعلق الأمر بـ I/O، وسيحتاج المُستدعي توضيحًا لهذه الأسباب. في تلك الحالات يكون نوع النتيجة الإضافية هو "الخطأ/error".

إن النوع المُدمج error هو من أنواع الواجهة "interface". سنرى المزيد من التفاصيل عما يعنيه هذا المصطلح و أثره على التعامل مع الأخطاء في الفصل السابع. الآن يكفي أن نعرف أن النوع error يُمكن أن يكون nil أو non-nil، وأن nil يدل على النجاح و non-nil يدل على الفشل، وأن error الـ non-nil يُظهر سلسلة رسالة خطأ يمكن أن نحصل عليها من خلال استدعاء طريقة Error أو طباعتها من خلال استدعاء fmt.Println(err) أو fmt.Printf("%v", err).

عندما تعيد وظيفة خطأ non-nil، فإن نتائجها الأخرى تكون غير مُعرّفة ويجب تجاهلها. بالرغم من أن القليل من الوظائف من الممكن أن تُرجع نتائج جزئية عند حدوث خطأ. على سبيل المثال، إذا حدث الخطأ أثناء القراءة من ملف، فإن استدعاء Read يعيد عدد البايتات التي تمكن من قراءتها، وقيمة خطأ تشرح المشكلة. قد يحتاج بعض المُستدعين لمعالجة البيانات الناقصة قبل التعامل مع الخطأ من أجل تصحيح السلوك، لذلك من المهم أن توثق هذه الوظائف نتائجها بوضوح.

إن منهج Go يميزها عن العديد من اللغات الأخرى التي تقدم تقرير بالخطأ باستخدام الاستثناءات وليس القيم الاعتيادية، وبالرغم من أن Go بها آلية استثناء نوعًا ما، كما سنرى في القسم 5.9، إلا أنها تُستخدم فقط في تقديم تقارير عن الأخطاء غير المتوقعة حقًا والتي تشير إلى وجود خلل، وليس الأخطاء الروتينية التي يتوقعها المكتوب بطريقة احترافية.

إن سبب هذا التصميم هو أن الاستثناءات تميل لخلط وصف الخطأ مع تدفق التحكم المطلوب لمعالجة الخطأ، مما يؤدي عادة إلى ناتج غير مرغوب فيه: تُقدم أخطاء روتينية إلى المستخدم النهائي في شكل أثر رصة غير مفهوم، وممتلئ بمعلومات حول بنية البرنامج ولكن ينقصه السياق المفهوم الذي يوضح الخطأ الذي وقع.

على النقيض، تستخدم Go آليات تدفق تحكم (control-flow) اعتيادية مثل if و return للاستجابة للأخطاء. يتطلب هذا الأسلوب دون شك توجيه المزيد من الانتباه لمنطق التعامل مع الخطأ، ولكن هذا هو الهدف منه بالضبط.

5.4.1 استراتيجيات التعامل مع الأخطاء

عندما يعيد استدعاء الوظيفة خطأ، فإن مسؤولية المستدعي هي التحقق منه واتخاذ الإجراءات المناسبة للتعامل معه. سيكون هناك احتمالات مختلفة تبعًا لكل موقف. دعنا نلقي نظرة على خمسة منهم.

الاحتمال الأول والأكثر شيوعًا هو نقل الخطأ، بحيث يصبح الفشل في روتين فرعي فشلًا في روتين الاستدعاء. لقد رأينا أمثلة على هذا في وظيفة findLinks في القسم 5.3. لو فشل الاستدعاء لـ http.Get، فإن findLinks ستعيد خطأ HTTP للمستدعي دون مزيد من الشوشرة:

```
resp, err := http.Get(url)
```

```
if err != nil {
    return nil, err
}
```

على النقيض، لو فشل استدعاء html.Parse، فإن findLinks لا تعيد خطأ محلل HTML بشكل مباشر لأنها ينقصها لمعلومتين ضروريتين وهما: أن الخطأ وقع في المحلّل، و URL المستند الذي كان قيد التحليل. في هذه الحالة، تبني findLinks رسالة خطأ جديدة تتضمن كلتا المعلومتين وكذلك خطأ التحليل الضمني:

```
doc, err := html.Parse(resp.Body)
resp.Body.Close()
if err != nil {
    return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
}
```

تُهيئ وظيفة fmt.Errorf رسالة الخطأ باستخدام fmt.Sprintf، وتعيد قيمة خطأ جديدة. نحن نستخدمها لبناء أخطاء وصفية من خلال إلحاق معلومات سياق إضافية برسالة الخطأ الأصلية بشكل تتابعي. عندما تعالج الوظيفة الرئيسية للبرنامج الخطأ في النهاية، فإنها يجب أن تقدم سلسلة سببية واضحة تبدأ من جذر المشكلة وتنتهي بالفشل النهائي، وهو ما يُذكّرنا بتحقيقات وكالة ناسا في الحوادث التي تقع لديها:

```
genesis: crashed: no parachute: G-switch failed: bad relay orientation
```

نظرًا لكون رسائل الخطأ ترتبط معًا عادة في سلسلة واحدة، فلا يجب أن نكتب سلاسل الرسالة بحروف كبيرة، ويجب تجنب الأسطر الجديدة. قد تكون الأخطاء الناتجة طويلة، ولكنها ستكون مستقلة وقائمة بذاتها عندما تجدها أدوات مثل grep.

اهتم بالتفاصيل جدًا عند تصميم رسائل الخطأ، بحيث تقدم كل رسالة وصفا واضحا للمشكلة بتفاصيل مهمة وكافية، وكن متسقًا أيضًا، بحيث يُعاد الخطأ بواسطة نفس الوظيفة أو مجموعة من الوظائف المتشابهة في الشكل والتي تنتمي لنفس الحزم، ويمكن التعامل معها بنفس الطريقة.

على سبيل المثال، تضمن حزمة os أن كل خطأ يُعاد عن طريقة عملية ملف، مثل os.Open، أو طُرق Read و Write أو Close الخاصة بالملف المفتوح، لا تصف طبيعة الفشل فقط (الدخول غير مسموح، هذا المسار غير موجود، إلخ)، ولكن تذكر أيضًا اسم الملف، بحيث لا يحتاج المستدعي لتضمين تلك المعلومة في رسالة الخطأ التي يبنيها.

بشكل عام، الاستدعاء f(x) مسؤول عن تقديم تقرير عن محاولة القيام بالعملية f، وقيمة المعطى x، المرتبطين بسياق الخطأ. إن المستدعي مسؤول عن إضافة المعلومات الإضافية التي يمتلكها، ولكن الاستدعاء f(x) غير مسؤول عن هذا، كما يتضح من عنوان URL الموجود في استدعاء html.Parse أعلاه.

لننتقل الآن إلى الاستراتيجية الثانية في التعامل مع الأخطاء. بالنسبة للأخطاء التي تمثل مشكلات عابرة وغير متوقعة، قد يكون من المنطقي "إعادة محاولة" تنفيذ العملية التي فشلت سابقًا، مع وضع مدة مناسبة بين المحاولات، وربما تجربة عدد محدود من المحاولات، أو تحديد وقت معين للمحاولة قبل التخلي عن الأمر بشكل نهائي.

```
gopl.io/ch5/wait
// WaitForServer attempts to contact the server of a URL.
// It tries for one minute using exponential back-off.
// It reports an error if all attempts fail.
func WaitForServer(url string) error {
    const timeout = 1 * time.Minute
    deadline := time.Now().Add(timeout)
    for tries := 0; time.Now().Before(deadline); tries++ {
        _, err := http.Head(url)
        if err == nil {
            return nil // success
        }
        log.Printf("server not responding (%s); retrying...", err)
        time.Sleep(time.Second << uint(tries)) // exponential back-off
    }
    return fmt.Errorf("server %s failed to respond after %s", url, timeout)
}
```

ثالثًا، لو كان التقدم مستحيلًا، يمكن للمستدعي طباعة الخطأ وإيقاف البرنامج بسلاسة، ولكن يجب أن يظل هذا الإجراء مقصورًا على الحزمة الأساسية للبرنامج بشكل عام. يجب على وظائف المكتبة عادة نقل الأخطاء للمستدعي، ما لم يكن الخطأ إشارة على تضارب داخلي، أي علة برمجية:

```
// (In function main.)
if err := WaitForServer(url); err != nil {
    fmt.Fprintf(os.Stderr, "Site is down: %v\n", err)
    os.Exit(1)
}
```

إن أحد الطرق الملائمة الأخرى لتحقيق نفس التأثير هي استدعاء log.Fatalf، والتي ستُلحق الوقت والتاريخ برسالة الخطأ كما هي العادة في كل وظائف log:

```
if err := WaitForServer(url); err != nil {
    log.Fatalf("Site is down: %v\n", err)
}
```

إن الصيغة الافتراضية مفيدة في الخادم الذي يعمل لفترة طويلة، ولكنها أقل فائدة في الأدوات التفاعلية:

```
2006/01/02 15:04:05 Site is down: no such domain: bad.gopl.io
```

لو أردنا الوصول لناتج أكثر جاذبية، يمكننا وضع سابقة تستخدمها حزمة log لتسمية الأمر، ومنع عرض التاريخ والوقت:

```
log.SetPrefix("wait: ")
log.SetFlags(0)
```

رابعًا، قد يكون كافيًا في بعض الحالات تسجيل الخطأ، ثم المتابعة، وإن كان هذا يمكن أن يقلل مستوى الأداء قليلًا. مرة أخرى، يوجد إمكانية للاختيار بين استخدام حزمة log، التي تضيف السابقة المعتادة:

```
if err := Ping(); err != nil {
    log.Printf("ping failed: %v; networking disabled", err)
}
```

وبين الطباعة مباشرة على تيار الخطأ القياسي:

```
if err := Ping(); err != nil {
    fmt.Fprintf(os.Stderr, "ping failed: %v; networking disabled\n", err)
}
```

(تُلحق كل وظائف log بسطر جديد لو لم يكن هناك واحد موجود بالفعل).

وخامسًا وأخيرًا، يمكننا – في الحالات النادرة – تجاهل الخطأ تمامًا وسيكون هذا آمنًا:

```
dir, err := ioutil.TempDir("", "scratch")
if err != nil {
    return fmt.Errorf("failed to create temp dir: %v", err)
}
// ... use temp dir...
os.RemoveAll(dir) // ignore errors; $TMPDIR is cleaned periodically
```

إن استدعاء os.RemoveAll قد يفشل، ولكن البرنامج يتجاهله لأن نظام التشغيل ينظم المسار المؤقت دوريًا. في هذه الحالة، كان التخلص من الخطأ متعمدا، ولكن منطق البرنامج كان سيطبق نفس الإجراء لو كنا نسينا التعامل معه. اعتَد على دراسة الأخطار بعد كل استدعاء للوظيفة، وعندما تتجاهل خطأ عمدًا، وثق قصدك بوضوح.

إن التعامل مع الأخطاء في Go له إيقاع محدد، فبعد التحقق من الخطأ، يتم التعامل مع الفشل قبل النجاح عادة. لو تسبب الفشل في عودة وظيفة، فإن منطق النجاح لا يُزاح داخل كتلة أخرى، ولكنه يتبع نفس المسار ولكن على المستوى الخارجي. تُظهر الوظائف عادة بنية مشتركة، مع سلسلة من الفحوص المبدئية لرفض الأخطاء، ويليها جوهر الوظيفة في النهاية، والذي يُزاح بأدنى درجة ممكنة.

5.4.2 نهاية الملف (EOF) – End of File (EOF)

إن مجموعة الأخطاء المتنوعة التي يمكن أن تعيدها الوظيفة قد تثير اهتمام المستخدم النهائي، ولكنها مزعجة بالنسبة لمنطق البرنامج المتضمن فيها، لكن من حيث لآخر، قد يضطر البرنامج لاتخاذ إجراءات مختلفة بناء على نوع الخطأ الذي

وقع. انظر محاولة قراءة n من بايتات البيانات من ملف. لو كانت n هي طول الملف، فإن أي خطأ يمثل فشلًا. من ناحية أخرى، لو حاول المستدعي بشكل متكرر قراءة أجزاء ذات حجم ثابت حتى استنزاف الملف بأكمله، فإن المستدعي يجب أن يستجيب لحالة نهاية الملف بشكل مختلف عن استجابته لكل أنواع الأخطاء الأخرى. لهذا السبب، تضمن حزمة io أن أي فشل في القراءة ناتج عن حالة "نهاية الملف" سيظهر في صورة خطأ مميز هو io.EOF، والذي يُعرّف كالتالي:

```
package io
import "errors"
// EOF is the error returned by Read when no more input is available.
var EOF = errors.New("EOF")
```

يمكن للمستدعي كشف هذه الحالة باستخدام مقارنة بسيطة، كما هو موضح في الحلقة أدناه، والتي تقرأ runes من مُدخل قياسي. (يقدم برنامج charcount في القسم 4.3 مثالا معقدا أكثر).

```
in := bufio.NewReader(os.Stdin)
for {
	r, _, err := in.ReadRune()
	if err == io.EOF {
		break // finished reading
	}
	if err != nil {
		return fmt.Errorf("read failed: %v", err)
	}
	// ...use r...
}
```

نظرًا لأنه لا يوجد معلومات في حالة "نهاية الملف" لتقديم تقرير عنها إلا حقيقة انتهاء الملف، تقدم io.EOF رسالة خطأ ثابتة هي "EOF". قد نحتاج مع الأخطاء الأخرى إلى تقديم تقرير بجودة وكمية الخطأ، وبالتالي فإن قيمة الخطأ الثابتة لن تكون كافية هنا. سنقدم في القسم 7.11 طريقة منهجية أكثر لتمييز قيم خطأ معينة عن غيرها.

5.5 قيم الوظيفة - Function Vaues

إن الوظائف هي قيم من الدرجة الأولى (first-class values) في لغة جو: إن قيم الوظيفة لها أنواع مثلها مثل القيم الأخرى، ويمكن نسبتها إلى متغيرات أو تمريرها إلى وظائف أو إعادتها منها. يمكن استدعاء قيمة الوظيفة مثلها مثل أي وظيفة أخرى. كمثال:

```
func square(n int) int     { return n * n }
func negative(n int) int   { return -n }
func product(m, n int) int{ return m * n }
```

```
f := square
fmt.Println(f(3)) // "9"

f = negative
fmt.Println(f(3)) // "-3"
fmt.Printf("%T\n", f) // "func(int) int"
f = product // compile error: can't assign f(int, int) int to f(int) int
```

إن القيمة الصفرية لنوع وظيفة هي nil. واستدعاء قيمة وظيفة قيمتها nil قد يسبب هلع:

```
var f func(int) int
f(3) // panic: call of nil function
```

يمكن مقارنة قيم الوظيفة مع nil:

```
var f func(int) int
if f != nil {
    f(3)
}
```

ولكنها غير قابلة للمقارنة، وبالتالي لا يمكن مقارنتها ببعضها أو استخدامها كمفاتيح في خريطة.

تساعدنا قيم الوظيفة على وضع معاملات لوظائفنا، معاملات للسلوك وليس البيانات فقط. تحتوي المكتبات القياسية على العديد من الأمثلة. كمثال، تطبق strings.Map وظيفة على كل حرف في سلسلة، وتدمج النتائج معًا لتكوين سلسلة أخرى.

```
func add1(r rune) rune { return r + 1 }
fmt.Println(strings.Map(add1, "HAL-9000")) // "IBM.:111"
fmt.Println(strings.Map(add1, "VMS"))//    "WNT"

fmt.Println(strings.Map(add1, "Admix")) // "Benjy"
```

تستخدم وظيفة findLinks من القسم 5.2 وظيفة مساعدة، visit و، لزيارة كل العقد في مستند HTML وتطبيق إجراء على كل واحد منها. لو استخدمنا قيمة الوظيفة، سنتمكن من فصل منطق اجتياز الشجرة عن منطق الإجراء، وتطبيقه على كل عقدة، مما يُمكننا من إعادة استخدام الاجتياز مع إجراءات مختلفة.

```
gopl.io/ch5/outline2
// forEachNode calls the functions pre(x) and post(x) for each node
// x in the tree rooted at n. Both functions are optional.
// pre is called before the children are visited (preorder) and
// post is called after (postorder).
func forEachNode(n *html.Node, pre, post func(n *html.Node)) {
   if pre != nil {
      pre(n)
   }
```

```
    for c := n.FirstChild; c != nil; c = c.NextSibling {
        forEachNode(c, pre, post)
    }
    if post != nil {
        post(n)
    }
}
```

تقبل وظيفة forEachNode معطيات وظيفتين، واحدة تستدعيها قبل زيارة أطفال العقدة، والأخرى تستدعيها بعدها. يمنح هذا الترتيب للمستدعي مرونة كبيرة. كمثال، تطبع الوظائف startElement و endElement وسوم بداية ونهاية عنصر HTML مثل ...:

```
var depth int
func startElement(n *html.Node) {
    if n.Type == html.ElementNode {
        fmt.Printf("%*s<%s>\n", depth*2, "", n.Data)
        depth++
    }
}
func endElement(n *html.Node) {
    if n.Type == html.ElementNode {
        depth--
        fmt.Printf("%*s</%s>\n", depth*2, "", n.Data)
    }
}
```

تزيج الوظائف الناتج أيضًا باستخدام خدعة fmt.Printf، بينما أن ظرف * في %*s يطبع سلسلة مبطنة بعدد متباين من المسافات. تقدم المعطيات depth*2 و "" كل من العرض والسلسلة.

لو استدعينا forEachNode لمستند HTML، كالتالي:

```
forEachNode(doc, startElement, endElement)
```

سنحصل على تباين أكثر تفصيلًا لناتج برنامج المخطط السابق الخاص بنا:

```
$ go build gopl.io/ch5/outline2
$ ./outline2 http://gopl.io <html>
<head>
    <meta>
    </meta>
    <title>
    </title>
    <style>
    </style>
    </head>
        <body>
            <table>
                <tbody>
```

```
                    <tr>
                        <td>
                            <a>
                                <img>
                                </img>
...
```

تمرين 5.7: طوّر startElement و endElement إلى طابعة HTML جيدة. اطبع عقد التعليق، وعقد النص، وصفات كل عنصر (<a href='...'>). استخدم اختصار لـ ms مثل </img> بدلًا من <img></img> عندما لا يكون للعنصر أطفال. اكتب اختبار لضمان تحليل الناتج بنجاح. (انظر الفصل 11).

تمرين 5.8: عدّل كل forEachNode بحيث تعيد وظائف pre و post (أو قبل وبعد) نتيجة قيمة منطقية (بوليان) تشير إلى ما إذا كنا سنواصل الاجتياز أم لا. استخدمها لكتابة وظيفة ElementByID بالتوقيع التالي الذي يجد أول عنصر HTML ذو صفة id المحددة. يجب أن توقف الوظيفة الاجتياز بمجرد إيجاد نتيجة مطابقة.

```
func ElementByID(doc *html.Node, id string) *html.Node
```

تمرين 5.9: اكتب وظيفة expand(s string, f func(string) string) string تحل محل كل سلسلة فرعية "$foo" داخل s بواسطة نص تعيده f("foo").

5.6 الوظائف المجهولة - Anonymous Functions

يمكن إعلان الوظائف المسماة على مستوى الحزمة فقط، ولكن يمكننا استخدام "حروف الوظيفة" (function literal) للدلالة على قيمة وظيفة داخل أي تعبير. يُكتب حرف الوظيفة كإعلان الوظيفة، ولكن بدون اسم بعد كلمة func المفتاحية. إنه تعبير تُستدعى قيمته كوظيفة مجهولة (anonymous function).

تتركنا حروف الوظيفة نُعرّف الوظيفة عند نقطة استخدامها. كمثال، يمكن إعادة الاستدعاء السابق لـ strings.Map كالتالي:

```
strings.Map(func(r rune) rune { return r + 1 }, "HAL-9000")
```

الأهم من ذلك، أن الوظائف المُعرّفة بهذه الطريقة تستطيع الدخول للبيئة اللغوية بأكملها، وبالتالي يمكن للوظيفة الداخلية الرجوع لمتغيرات من الوظيفة المغلّفة، كما يوضح المثال التالي:

```
gopl.io/ch5/squares
// squares returns a function that returns
// the next square number each time it is called.
```

```go
func squares() func() int {
    var x int
    return func() int {
        x++
        return x * x
    }
}
func main() {
    f := squares()
    fmt.Println(f()) // "1"
    fmt.Println(f()) // "4"
    fmt.Println(f()) // "9"
    fmt.Println(f()) // "16"
}
```

تعيد الوظيفة squares وظيفة أخرى من النوع func()int. يخلق استدعاء squares متغيرا محليا x ويعيد وظيفة مجهولة تزيد x وتعيد مربعه كلما تم استدعاءها. سينشأ عن الاستدعاء الثاني لـ squares متغير x ثاني ويعيد وظيفة مجهولة جديدة تزيد هذا المتغير.

يوضح مثال المربعات أن قيم الوظيفة ليست مجرد شفرة، ولكن يمكن أن يصبح لها حالة خاصة بها أيضًا. يمكن الوصول للوظيفة الداخلية المجهولة وتحديث المتغيرات الموضعية لمربعات الوظيفة المغلّفة. إن مراجع المتغير المُخبأ هذه هي سبب تصنيفنا للوظائف كأنواع مرجعية، وسبب عدم قابلية قيم الوظيفة للمقارنة. إن قيم الوظيفة من هذا النوع تُطبق باستخدام تقنية يُطلق عليها الإغلاقات (closures)، ويستخدم مُبرمجو Go هذا المصطلح عادة للإشارة لقيم الوظيفة.

نرى هنا مرة أخرى مثالًا لا يتحدد فيه عُمر المتغير وفقًا لنطاقه: يتواجد المتغير x بعد أن عادت المربعات داخل main، بالرغم من أن x مختبئ داخل f.

إن أحد الأمثلة الأكاديمية نوعًا ما على الوظائف المجهولة، هي مشكلة حساب تسلسل مقررات علوم الكمبيوتر بحيث يحقق كل منها المتطلبات المسبقة الخاصة بكل واحدة منها. إن المتطلبات المسبقة مُقدّمة في جدول الطلبات المسبقة (أو جدول prereqs) أدناه، وهو مخطط خارج من كل مقرر إلى قائمة المقررات التي يجب إكمالها قبله.

```go
gopl.io/ch5/toposort
// prereqs maps computer science courses to their prerequisites.
var prereqs = map[string][]string{
    "algorithms": {"data structures"},
    "calculus":   {"linear algebra"},
    "compilers": {
        "data structures",
        "formal languages",
        "computer organization",
    },
    "data structures":       {"discrete math"},
    "databases":             {"data structures"},
    "discrete math":         {"intro to programming"},
    "formal languages":      {"discrete math"},
    "networks":              {"operating systems"},
```

```
    "operating systems":     {"data structures", "computer organization"},
    "programming languages": {"data structures", "computer organization"},
}
```

يُعرف هذا النوع من المشكلات باسم الفرز الطوبولوجي. تشكل المعلومات المطلوبة مسبقًا رسم بياني موجه ذو عقدة مناظرة لكل مقرر، وتتجه من كل مقرر إلى المقرر التالي الذي يعتمد عليها. إن الرسم البياني غير دوري، فلا يوجد مسار من مقرر يؤدي للعودة لنفس المقرر في النهاية. يمكننا حساب تسلسل سليم باستخدام البحث بالعمق أولًا في الرسم البياني باستخدام الشفرة أدناه:

```
func main() {
    for i, course := range topoSort(prereqs) {
        fmt.Printf("%d:\t%s\n", i+1, course)
    }
}
func topoSort(m map[string][]string) []string {
    var order []string
    seen := make(map[string]bool)
    var visitAll func(items []string)
    visitAll = func(items []string) {
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                visitAll(m[item])
                order = append(order, item)
            }
        }
    }
    var keys []string
    for key := range m {
        keys = append(keys, key)
    }
    sort.Strings(keys)
    visitAll(keys)
    return order
}
```

عندما تحتاج وظيفة مجهولة للتكرار، كما هو الحال في هذا المثال، يجب أن نعلن عن متغير أولا، ثم نخصص وظيفة مجهولة لهذا المتغير. لو دُمجت تلك الخطوتين في الإعلان، فإن حرف الوظيفة لن يكون داخل نطاق المتغير visitAll، وبالتالي لن يكون هناك أي وسيلة تُمكنه من استدعاء نفسه بشكل تكراري:

```
visitAll := func(items []string) {
    // ...
    visitAll(m[item]) // compile error: undefined: visitAll
    // ...
}
```

موضح أدناه ناتج برنامج toposort، إنه خاصية محددة ومرغوبة عادة، ولكنها ليست مجانية دائمًا. إن قيم خريطة prereqs هنا هي شرائح، وليس مزيد من الخرائط، وبالتالي فإن ترتيب تكرارهم محدد، وقد فرزنا كل مفاتيح prereqs قبل عمل الاستدعاءات الأولية لـ visitAll.

```
1:  intro to programming
2:  discrete math
3:  data structures
4:  algorithms
5:  linear algebra
6:  calculus
7:  formal languages
8:  computer organization
9:  compilers
10: databases
11: operating systems
12: networks
13: programming languages
```

لنعود الآن إلى مثال findLinks. لقد نقلنا وظيفة استخلاص الرابط links.Extract إلى حزمتها الخاصة، حيث أننا سنستخدمها مرة أخرى في الفصل الثامن. وقد استبدال وظيفة visit بوظيفة مجهولة تلتحق بشريحة الروابط بشكل مباشر، واستخدمنا forEachNode للتعامل مع الاجتياز. كما مررنا nil كمُعطى بَعدي (post) نظرًا لأن الاستخلاص يحتاج لوظيفة قبلية (pre) فقط.

```
gopl.io/ch5/links
import (
    "fmt"
    "net/http"
    "golang.org/x/net/html"
)
// Extract makes an HTTP GET request to the specified URL, parses
// the response as HTML, and returns the links in the HTML document.
func Extract(url string) ([]string, error) {
    resp, err := http.Get(url)
    if err != nil {
        return nil, err
    }
    if resp.StatusCode != http.StatusOK {
        resp.Body.Close()
        return nil, fmt.Errorf("getting %s: %s", url, resp.Status)
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return nil, fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    var links []string
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "a" {
```

```
            for _, a := range n.Attr {
                if a.Key != "href" {
                    continue
                }
                link, err := resp.Request.URL.Parse(a.Val)
                if err != nil {
                    continue // ignore bad URLs
                }
                links = append(links, link.String())
            }
        }
    }
    forEachNode(doc, visitNode, nil)
    return links, nil
}
```

بدلًا من إلحاق قيمة صفة href الخام بشريحة الروابط، تحللها تلك النسخة كـ URL مقارنة بـ URL الأساسي الخاص بالمستند، resp.Request.URL. إن الرابط الناتج هو الشكل النهائي، والمناسب للاستخدام في استدعاء http.Get.

إن الزحف عبر الويب هو في الأساس مشكلة اجتياز الرسم البياني. وقد أوضح مثال topoSort اجتياز بالعمق أولا. سنستخدم في زاحف الويب الخاص بنا اجتياز بالعرض أولا، مبدئيًا على الأقل. سنستكشف في الفصل الثامن الاجتياز المتزامن.

تُغلِف الوظيفة أدناه جوهر الاجتياز بالعرض أولا، ويقدم المستدعي قائمة عمل مبدئية بالعناصر التي ستتم زيارتها، وقيمة الوظيفة f لاستدعاء كل عنصر. يُحدد كل عنصر بواسطة سلسلة، وتعيد الوظيفة f قائمة بالعناصر الجديدة التي ستُلحق بقائمة العمل. تعود وظيفة breadthFirst عند زيارة كل العناصر، وهي تحافظ على سلسلة من السلاسل لضمان عدم زيارة عنصر مرتين.

```
gopl.io/ch5/findlinks3
// breadthFirst calls f for each item in the worklist.
// Any items returned by f are added to the worklist.
// f is called at most once for each item.
func breadthFirst(f func(item string) []string, worklist []string) {
    seen := make(map[string]bool)
    for len(worklist) > 0 {
        items := worklist
        worklist = nil
        for _, item := range items {
            if !seen[item] {
                seen[item] = true
                worklist = append(worklist, f(item)...)
            }
        }
    }
}
```

كما شرحنا بشكل عابر في الفصل الثالث، فإن المُعطى ''f(item)...'' سيجعل كل العناصر الموجودة في القائمة - والتي أعادتها f – تُلحَق بقائمة العمل.

إن العناصر في زاحفنا هي URLs، إن وظيفة crawl التي سنقدمها لـ breadthFirst ستطبع الـ URL، وتستخلص روابطها، وتعيدها بحيث يتم زيارتها هي أيضًا.

```
func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

سنستخدم معطيات سطر الأوامر كـ URLs مبدئية كي نبدأ عمل الزاحف:

```
func main() {
    // Crawl the web breadth-first,
    // starting from the command-line arguments.
    breadthFirst(crawl, os.Args[1:])
}
```

لنزحف عبر الويب بداية من https://golang.org، وستجد في الجزء التالي بعض الروابط الناتجة:

```
$ go build gopl.io/ch5/findlinks3
$ ./findlinks3
https://golang.org
https://golang.org/
https://golang.org/doc/
https://golang.org/pkg/
https://golang.org/project/
https://code.google.com/p/go-tour/
https://golang.org/doc/code.html
https://www.youtube.com/watch?v=XCsL89YtqCs
http://research.swtch.com/gotour
https://vimeo.com/53221560
```

تنتهي العملية عند الزحف على كل صفحات الويب التي يمكن الوصول إليها أو استنزاف ذاكرة الكمبيوتر.

تمرين 5.10: أعِد كتابة topoSort بحيث تستخدم الخرائط بدلًا من الشرائط، وتحذف الفرز المبدئي. تحقق من أن النتائج هي ترتيبات طوبولوجية صحيحة، حتى وإن كانت غير محددة.

تمرين 5.11: يقرر مدرس مقرر الجبر الخطي أن التفاضل والتكامل مطلب مسبق ضروري. وسّع وظيفة topoSort لتقدم تقرير بالدورات.

تمرين 5.12: تشترك الوظائف startElement و startElement في gopl.io/ch5/outline2 (انظر 5.5) في متغير عام هو العمق. حوّلهم إلى وظائف مجهولة تشترك في متغير محلي في وظيفة المخطط.

تمرين 5.13: عدّل الزحف بحيث يصنع نسخ محلية من الصفحات التي يجدها، ويُنشئ مجلدات (directories) عند الضرورة. لا تصنع نسخ من الصفحات الآتية من نطاق مختلف. كمثال، لو أتت الصفحة الأصلية من golang.com، فاحفظ كل الملفات الموجودة فيها، ولكن استبعد الملفات القادمة من vimeo.com.

تمرين 5.14: استخدم وظيفة breadthFirst لاستكشاف بنية مختلفة. كمثال، يمكنك استخدام اعتمادات المقرر من مثال topoSort (الرسم البياني الموجه)، أو التسلسل الهرمي لنظام الملفات على جهاز الكمبيوتر (شجرة)، أو قائمة بطرق الحافلات أو المترو المُحملة من موقع حكومة مدينتك (رسم بياني غير موجه).

5.6.1 تحذير: التقاط متغيرات التكرار - Caveat: Capturing Iteration Variables

سننظر في هذا الفصل إلى شِرك قواعد النطاق اللغوي الخاصة بـ Go، والتي يمكن أن تسبب نتائج مفاجئة. نحن نحثك على فهم المشكلة قبل المتابعة، لأن هذا الشرك قد يقع فيه المبرمجين الخبراء حتى.

فكر في برنامج يجب أن ينشئ مجموعة من المجلدات ثم يحذفها لاحقًا. يمكننا استخدام شريحة من قيم الوظيفة لتحمل عمليات التنظيف. (للاختصار، مسحنا كل طرق معالجة الخطأ في هذا المثال).

```
var rmdirs []func()
for d := range tempDirs() {
	dir := d                  // NOTE: necessary!
	os.MkdirAll(dir, 0755) // creates parent directories too
	rmdirs = append(rmdirs, func() {
		os.RemoveAll(dir)
	})
}
// ...do some work...
for _, rmdir := range rmdirs {
	rmdir() // clean up
}
```

قد تتساءل، لماذا نخصص متغير الحلقة d إلى المتغير المحلي الجديد dir داخل جسم الحلقة بدلًا من مجرد تسمية متغير الحلقة dir كما نفعل في التنويع التالي غير الدقيق إلى حد كبير:

```
var rmdirs []func()
for _, dir := range tempDirs() {
	os.MkdirAll(dir, 0755)
	rmdirs = append(rmdirs, func() {
		os.RemoveAll(dir) // NOTE: incorrect!
	})
}
```

إن السبب هو أحد تبعات قواعد نطاق متغيرات الحلقة. سنجد في البرنامج الموضح أعلاه أن حلقة for تقدم كتلة لغوية جديدة يُعلن فيها المتغير dir، وأن كل قيم الوظيفة الناشئة عن تلك الحلقة "تلتقط" وتشارك نفس المتغير – موقع تخزين قابل للتوجيه، وليس قيمته في تلك اللحظة المحددة. تُحدث قيمة dir في التكرارات المتتابعة، وبالتالي بحلول وقت استدعاء وظائف التنظيف، يكون متغير dir حُدّث عدة مرات بواسطة حلقة for التي اكتملت الآن. من ثم، يحمل dir قيمة التكرار الأخير، وبالتالي ستحاول كل استدعاءات os.RemoveAll حذف نفس المجلد.

إن المتغير الداخلي المُقدم لمحاولة العمل والتحايل على هذه المشكلة – وهو dir في حالتنا – عادة ما يُمنح نفس اسم المتغير الخارجي المنسوخ منه، مما يؤدي إلى إعلانات متغير غريبة الشكل ولكنها ضرورية، كالتالي:

```
for _, dir := range tempDirs() {
    dir := dir // declares inner dir, initialized to outer dir
    // ...
}
```

إن الخطر ليس مقصورًا على حلقات for القائمة على النطاق، فالحلقة في المثال أدناه تعاني من نفس المشكلة بسبب الالتقاط غير المقصود لمتغير الفهرس i.

```
var rmdirs []func()
dirs := tempDirs()
for i := 0; i < len(dirs); i++ {
    os.MkdirAll(dirs[i], 0755) // OK
    rmdirs = append( rmdirs, func() {
        os.RemoveAll(dirs[i]) // NOTE: incorrect!
    })
}
```

تُواجه مشكلة التقاط متغير التكرار عادة عند استخدام عبارة go (الفصل الثامن) أو في defer (والتي سنراها خلال لحظات) حيث أن كلاهما يؤجل تنفيذ قيمة الوظيفة إلى ما بعد انتهاء الحلقة. لكن المشكلة ليست لازمة أو متأصلة في go أو defer.

5.7 الوظائف المتغيرة - Variadic Functions

إن الوظيفة المتغيرة (variadic function) هي وظيفة يمكن استدعاءها بعدد متباين من المعطيات، والأمثلة الأشهر هي fmt.Printf وتنويعاتها. تحتاج Printf إلى معطى واحد ثابت في البداية، ثم تقبل أي عدد من المعطيات اللاحقة.

يتطلب الإعلان عن الوظيفة المتغيرة أن يسبق نوع المعامل النهائي علامات القطع ''...''، والتي تشير إلى أن الوظيفة يمكن استدعاءها بأي عدد معطيات من هذا النوع.

```
gopl.io/ch5/sum
func sum(vals ...int) int {
    total := 0
    for _, val := range vals {
        total += val
    }
    return total
}
```

تعيد وظيفة sum أعلاه مجموعة معطيات int الصفرية أو الأكبر. إن نوع vals هو شريحة int[] داخل جسم الوظيفة، وعند استدعاء sum، يمكن تقديم أي عدد من القيم لمعامل vals الخاص بها.

```
fmt.Println(sum())           // "0"
fmt.Println(sum(3))          // "3"
fmt.Println(sum(1, 2, 3, 4)) // "10"
```

يخصص المستدعي ضمنيًا مصفوفة، وينسخ المعطيات فيها، ويمرر شريحة بالمصفوفة بأكملها إلى الوظيفة. من ثم، يعمل الاستدعاء الأخير أعلاه بنفس طريقة الاستدعاء أدناه، وهو ما يوضح كيفية استدعاء وظيفة متغيرة عندما تكون المعطيات موجودة في شريحة بالفعل: ضع علامات قطع بعد المعطى الأخير.

```
values := []int{1, 2, 3, 4}
fmt.Println(sum(values...)) // "10"
```

بالرغم من أن معامل int... يعمل كشريحة داخل جسم الوظيفة، إلا أن نوع الوظيفة المتغيرة مميز عن نوع الوظيفة الموجودة داخل معامل شريحة عادي.

```
func f(...int) {}
func g([]int) {}
fmt.Printf("%T\n", f) // "func(...int)"
fmt.Printf("%T\n", g) // "func([]int)"
```

تُستخدم الوظائف المتغيرة عادة في تهيئة السلسلة، وتبني وظيفة errorf أدناه رسالة خطأ مُهيّئة مع رقم السطر في البداية. إن اللاحقة f هي أحد تقاليد التسمية المتبعة على نطاق واسع في الوظائف المتغيرة التي تقبل سلسلة تهيئة Printf-style.

```
func errorf(linenum int, format string, args ...interface{}) {
    fmt.Fprintf(os.Stderr, "Line %d: ", linenum)
    fmt.Fprintf(os.Stderr, format, args...)
    fmt.Fprintln(os.Stderr)
}
```

```
linenum, name := 12, "count"
errorf(linenum, "undefined: %s", name) // "Line 12: undefined: count"
```

إن النوع {}interface يعني أن هذه الوظيفة يمكن أن تقبل أي قيم على الإطلاق مقابل معطياتها النهائية، كما سنوضح في الفصل السابع.

تمرين 5.15: اكتب وظائف متغيرة max و min، والمناظرة لـ sum. ما الذي يجب على هذه الوظائف فعله عند استدعاءها بدون معطيات؟ اكتب التنويعات التي تحتاج لمعطى واحد على الأقل.

تمرين 5.16: اكتب نسخة متغيرة من strings.Join.

تمرين 5.17: اكتب وظيفة متغيرة ElementsByTagName تعيد كل العناصر المطابقة لأسماء شجرة عقدة HTML وصفر أو أكثر. إليك مثال على استدعائين:

```
func ElementsByTagName(doc *html.Node, name ...string) []*html.Node

images := ElementsByTagName(doc, "img")
headings := ElementsByTagName(doc, "h1", "h2", "h3", "h4")
```

5.8 استدعاءات الوظيفة المؤجلة - Deferred Function Calls

استخدمت أمثلة findLinks الخاصة بنا ناتج http.Get كمُدخل لـ html.Parse. سيعمل هذا بشكل جيد لو كان محتوى URL المطلوب هو HTML، ولكن تحتوي العديد من الصفحات على صور، ونص بسيط، وصيغ ملفات أخرى. إن تغذية تلك الملفات لمحلل HTML قد يكون له آثار غير مرغوب فيها.

يجلب البرنامج أدناه مستند HTML ويطبع عنوانه. تفحص وظيفة title عنوان نوع المحتوى الموجود في استجابة الخام، وتعيد رسالة خطأ لو كان المستند ليس HTML.

```
gopl.io/ch5/titlel
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    // Check Content-Type is HTML (e.g., "text/html; charset=utf-8").
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        resp.Body.Close()
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
```

```
    }
    doc, err := html.Parse(resp.Body)
    resp.Body.Close()
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" &&
            n.FirstChild != nil {
            fmt.Println(n.FirstChild.Data)
        }
    }
    forEachNode(doc, visitNode, nil)
    return nil
}
```

هذه جلسة تقليدية عُدلت بشكل طفيف لتناسب ما نريد:

```
$ go build gopl.io/ch5/title1
$ ./titlel http://gopl.io
The Go Programming Language
$ ./titlel https://golang.org/doc/effective_go.html
Effective Go - The Go Programming Language
$ ./titlel https://golang.org/doc/gopher/frontpage.png
title: https://golang.org/doc/gopher/frontpage.png
has type image/png, not text/html
```

لاحظ استدعاء resp.Body.Close() المكرر، والذي يضمن أن title ستغلق اتصال الشبكة في كل مسارات التنفيذ، بما في ذلك حالات الفشل. مع زيادة تعقيد الوظائف، والاضطرار للتعامل مع المزيد من الأخطاء، يمكن أن يتحول تكرار منطق التنظيف إلى مشكلة صيانة. لنرى كيف تبسط آلية defer الجديدة في Go الأمور.

من الناحية التركيبية، تُعد عبارة defer استدعاء عادي لوظيفة أو طريقة يسبقه الكلمة الدلالية defer. تُقيّم تعبيرات الوظيفة والمعطى عند تنفيذ العبارة، ولكن الاستدعاء الفعلي "يؤجّل" حتى انتهاء الوظيفة التي تحتوي على عبارة التأجيل "defer"، سواء انتهت بطريقة عادية أو من خلال تنفيذ عبارة إعادة "return" أو الفشل في النهاية، أو في حالات استثنائية..لو حدث هلع. يمكن تأجيل أي عدد من الاستدعاءات، وأن تُنفذ بالترتيب العكسي للترتيب الذي أجلت به.

تُستخدم عبارة defer عادة مع العمليات المقترنة مثل الفتح والإغلاق، والاتصال وقطع الاتصال، أو الغلق والفتح، لضمان تحرير الموارد في كل الحالات، مهما كان تعقيد تدفق التحكم. إن المكان الصحيح لعبارة defer التي تحرر المورد هو بعد اكتساب المورد بنجاح مباشرةً. يحل استدعاء مؤجل منفرد محل كلا الاستدعائين السابقين لـ resp.Body.Close() في وظيفة title بسيطة:

```
gopl.io/ch5/title2
```

```
func title(url string) error {
    resp, err := http.Get(url)
    if err != nil {
        return err
    }
    defer resp.Body.Close()
    ct := resp.Header.Get("Content-Type")
    if ct != "text/html" && !strings.HasPrefix(ct, "text/html;") {
        return fmt.Errorf("%s has type %s, not text/html", url, ct)
    }
    doc, err := html.Parse(resp.Body)
    if err != nil {
        return fmt.Errorf("parsing %s as HTML: %v", url, err)
    }
    // ...print doc's title element...
    //!-
    visitNode := func(n *html.Node) {
        if n.Type == html.ElementNode && n.Data == "title" &&
            n.FirstChild != nil {
            fmt.Println(n.FirstChild.Data)
        }
    }
    forEachNode(doc, visitNode, nil)
    //!+
    return nil
}
```

يمكن استخدام نفس النمط مع الموارد الأخرى إضافة إلى اتصال الشبكة ، كمثال، لإغلاق ملف مفتوح:

```
io/ioutil
package ioutil
func ReadFile(filename string) ([]byte, error) {
     f, err := os.Open(filename)
     if err != nil {
           return nil, err
     }
     defer f.Close()
     return ReadAll(f)
}
```

أو لفتح mutex (انظر 9.2):

```
var mu sync.Mutex
var m = make(map[string]int)
func lookup(key string) int {
     mu.Lock()
     defer mu.Unlock()
     return m[key]
}
```

يمكن استخدام عبارة defer أيضًا لجمع إجراءات "عند الدخول/on entry" و"عند الخروج/on exit" معًا، عند تحليل الخلل في وظيفة معقدة. تستدعي وظيفة bigSlowOperation أدناه trace فورًا، والذي يقوم بإجراء "عند الدخول"، ثم

يعيد قيمة وظيفة تقوم بإجراء "عند الخروج" مناظر عند استدعاءها. إن تأجيل استدعاء الوظيفة المعادة بهذه الطريقة يُمكننا من بناء نقطة دخول وكل نقاط خروج الوظيفة في عبارة واحدة، وتمرير قيم – مثل وقت البدء – بين الإجزاءين، لكن لا تنسى الأقواس النهائية في عبارة defer، وإلا سيحدث إجراء "عند الدخول" بدلًا من ذلك عند الخروج، ولن يحدث إجراء "عند الخروج" على الإطلاق.

```
gopl.io/ch5/trace
func bigSlowOperation() {
    defer trace("bigSlowOperation")() // don't forget the extra parentheses
    // ...lots of work...
    time.Sleep(10 * time.Second) // simulate slow operation by sleeping
}
func trace(msg string) func() {
    start := time.Now()
    log.Printf("enter %s", msg)
    return func() { log.Printf("exit %s (%s)", msg, time.Since(start)) }
}
```

كلما استُدعيت bigSlowOperation، تسجل دخولها وخروجها والوقت المقضي بينهما. (استخدمنا time.Sleep لمحاكاة العملية البطيئة).

```
$ go build gopl.io/ch5/trace
$ ./trace
2015/11/18 09:53:26 enter bigSlowOperation
2015/11/18 09:53:36 exit bigSlowOperation (10.000589217s)
```

تعمل الوظائف المؤجلة "بعد" تحديث عبارات الإعادة "return" لمتغيرات نتيجة الوظيفة. ونظرًا لكون الوظيفة المجهولة يمكن أن تدخل لمتغيرات الوظيفة المغلفة لها، بما في ذلك النتائج المسماة، سنجد أن الوظيفة المجهولة المؤجلة يمكن أن تلاحظ نتائج الوظيفة.

فكر في الوظيفة double:

```
func double(x int) int {
     return x + x
}
```

يمكننا جعل الوظيفة تطبع معاملاتها ونتائجها كلما استدعيت من خلال تسمية متغيرها وإضافة عبارة defer.

```
func double(x int) (result int) {
     defer func() { fmt.Printf("double(%d) = %d\n", x, result) }()
     return x + x
}
_ = double(4)
// Output:
// "double(4) = 8"
```

إن هذه الخدعة تعتبر مبالغ فيها بالنسبة لوظيفة بسيطة مثل double، ولكنها قد تكون مفيدة في الوظائف ذات عبارات الإعادة المتعددة.

يمكن للوظيفة المجهولة المؤجلة تغيير القيم التي تعيدها الوظيفة المُغلفة للمستدعي:

```
func triple(x int) (result int) {
    defer func() { result += x }()
    return double(x)
}
fmt.Println(triple(4)) // "12"
```

نظرًا لكون الوظائف المؤجلة لا تُنفذ إلا في نهاية تنفيذ الوظيفة، سنجد أن عبارة defer في الحلقة بحاجة لمزيد من الفحص والتدقيق. يمكن أن تنفذ أوصاف الملف في الشفرة أدناه نتيجة عدم غلق أي ملف إلا بعد معالجة كل الملفات:

```
for _, filename := range filenames {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close() // NOTE: risky; could run out of file descriptors
    // ... process f...
}
```

إن أحد الحلول هو نقل جسد الحلقة، بما في ذلك عبارة defer، إلى وظيفة أخرى تُستدعى في كل تكرار.

```
for _, filename := range filenames {
    if err := doFile(filename); err != nil {
        return err
    }
}
func doFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    // ... process f...
}
```

إن المثال أدناه هو برنامج جلب (fetch) مُحسّن (انظر 1.5)، يكتب استجابة HTML لملف محلي بدلًا من الخرج القياسي، وهو يشتق اسم الملف من آخر مكون في مسار URL، والذي يحصل عليه باستخدام وظيفة path.Base.

```
gopl.io/ch5/fetch
// Fetch downloads the URL and returns the
// name and length of the local file.
func fetch(url string) (filename string, n int64, err error) {
```

```
    resp, err := http.Get(url)
    if err != nil {
        return "", 0, err
    }
    defer resp.Body.Close()
    local := path.Base(resp.Request.URL.Path)
    if local == "/" {
        local = "index.html"
    }
    f, err := os.Create(local)
    if err != nil {
        return "", 0, err
    }
    n, err = io.Copy(f, resp.Body)
    // Close file, but prefer error from Copy, if any.
    if closeErr := f.Close(); err == nil {
        err = closeErr
    }
    return local, n, err
}
```

إن الاستدعاء المؤجل لـ resp.Body.Close يجب أن يكون مألوفًا الآن. من المغري استخدام استدعاء مؤجل ثاني لـ f.Close لإغلاق ملف محلي، ولكن هذا سيكون خاطئا إلى حد ما، لأن os.Create تفتح ملف للكتابة، وتنشئه حسب الحاجة. لا يُقدم تقرير بأخطاء الكتابة بشكل فوري في العديد من نظم الملفات، خاصة NFS، ولكن يمكن تأجيل ظهورها حتى إغلاق الملف. إن الفشل في فحص نتيجة عملية الإغلاق يمكن أن يؤدي لفقدان بيانات كثيرة دون أن نلاحظ هذا. لكن لو فشل كل من io.Copy و f.Close، فسيكون من الأفضل لو قدمنا تقرير الخطأ من io.Copy لأنه حدث أولا، ومن المرجح أن يخبرنا بالسبب الجذري.

تمرين 5.18: أعِد كتابة وظيفة fetch بحيث تستخدم defer في إغلاق ملف قابل للكتابة عليه، دون تغيير سلوك الوظيفة.

5.9 الهلع - Panic

يلتقط نظام أنواع Go العديد من الأخطاء في وقت الترجمة، ولكن هناك أخطاء أخرى، مثل الدخول لمصفوفة خارج الحدود أو الوصول إلى مؤشر nil، تحتاج إلى فحص في زمن التشغيل للتأكد من وجودها. عندما يكشف زمن تشغيل Go تلك الأخطاء يصاب بحالة هلع.

يتوقف التنفيذ الطبيعي للبرنامج خلال حالة الهلع التقليدية، وتُنفذ كل استدعاءات الوظيفة المؤجلة في هذا الـ روتين-جو، وينهار البرنامج مع ظهور رسالة تسجيل. تتضمن رسالة التسجيل "قيمة الهلع" (panic value)، والتي عادة ما تكون رسالة خطأ من نوع ما، كما أن أثر الرصة "stack trace" الخاص بكل روتين-جو يظهر رصة من استدعاءات الوظيفة

التي كانت نشطة في وقت الهلع. تحتوي رسالة السجل على معلومات كافية عادة لتشخيص السبب الجذري للمشكلة بدون تشغيل البرنامج مرة أخرى، لذا يجب أن تظهر الرسالة دائمًا في تقرير الخلل الخاص بالبرنامج المصاب بالهلع.

لا تنبع كل حالات الهلع من زمن التشغيل، فوظيفة panic المدمجة يمكن استدعاءها مباشرة، وهي تقبل أي قيمة كمعطى. قد يكون الهلع أفضل شيء يمكن فعله عند حدوث موقف "مستحيل"، كمثال، عند وصول التنفيذ لحالة لا يمكن أن تحدث منطقيًا:

```
switch s := suit(drawCard()); s {
case "Spades":        // ...
case "Hearts":        // ...
case "Diamonds":      // ...
case "Clubs":         // ...
default:
    panic(fmt.Sprintf("invalid suit %q", s)) // Joker?
}
```

إنه من العادات الجيدة أن تفحص الشروط المسبقة للدالة، ولكن هذا من السهل عمله بإفراط. وما لم تقم بتقديم رسالة خطأ ذات معلومات جيدة أو اكتشاف الخطأ بشكل مسبق، فإنه لا توجد فائدة من فحص شروط يقوم وقت التشغيل بالتأكد منه عنك.

```
func Reset(x *Buffer) {
    if x == nil {
        panic("x is nil") // unnecessary!
    }
    x.elements = nil
}
```

بالرغم من أن آلية الهلع في Go تشبه الاستثناءات في اللغات الأخرى، إلا أن الموقع الذي يُستخدم فيه الهلع مختلف إلى حد كبير. نظرًا لأن الهلع يؤدي لانهيار البرنامج، سنجد أنه يُستخدم في الأخطاء الجسيمة فقط، مثل التناقض المنطقي في البرنامج، ويعتبر المبرمجون اليقظون أي انهيار دليلًا على وجود خلل في شفرتهم. إن الأخطاء "المتوقعة" في البرنامج القوي المكتوب جيدًا، مثل الأخطاء الناتجة عن مُدخل غير صحيح، أو تهيئة خاطئة، أو فشل في I/O، يجب التعامل معها جميعًا بحرص، ومن الأفضل التعامل معها باستخدام قيم error.

انظر الوظيفة regexp.Compile التي تترجم تعبيرا اعتياديا إلى شكل مطابق كفء، وهي تعيد خطأ error لو استُدعيت عن طريق نمط سيء التكوين، ولكن فحص الخطأ غير ضروري وعبء لا حاجة له لو كان المستدعي يعلم أن هذا الاستدعاء المحدد لا يمكن أن يفشل. في تلك الحالة، سيكون من المنطقي أن يتعامل المستدعي مع الخطأ من خلال الهلع، لأنه يعتقد أن الخطأ مستحيل.

إن معظم التعبيرات المنتظمة هي حروف في شفرة مصدر البرنامج، وتقدم حزمة regexp وظيفة التغليف regexp.MustCompile التي تقوم بهذا الفحص:

```
package regexp
func Compile(expr string) (*Regexp, error) { /* ... */ }
func MustCompile(expr string) *Regexp {
     re, err := Compile(expr)
     if err != nil {
           panic(err)
     }
     return re
}
```

إن وظيفة المغلِف تجعل من العملاء قادرين على بدء متغير على مستوى الحزمة بتعبير منتظم مترجم مثل:

```
var httpSchemeRE = regexp.MustCompile('Ahttps?:') // "http:" or "https:"
```

لا يجب استدعاء MustCompile بقيم مُدخل غير موثوق فيها بالطبع، وتُعَد سابقة Must هي تقليد تسمية شائع في الوظائف من هذا النوع، مثل template.Must في القسم 4.6.

عند حدوث الهلع، تعمل كل الوظائف المؤجلة بترتيب معكوس، وتبدأ بالتي تحتوي على أعلى وظيفة في الرصة، وتتابع طريقها وصولًا للوظيفة main، كما يوضح البرنامج أدناه:

```
gopl.io/ch5/defer1
func main() {
    f(3)
}
func f(x int) {
    fmt.Printf("f(%d)\n", x+0/x) // panics if x == 0
    defer fmt.Printf("defer %d\n", x)
    f(x - 1)
}
```

يطبع البرنامج الناتج القياسي التالي عند تشغيله:

```
f(3)
f(2)
f(1)
defer 1
defer 2
defer 3
```

يحدث الهلع أثناء استدعاء f(0)، ويؤدي لتنفيذ الاستدعاءات المؤجلة الثلاثة لـ fmt.Printf. ثم تنهي runtime البرنامج، وتطبع رسالة الخطأ، وتلقي بالرصة إلى تدفق الخطأ القياسي (المبسط للتوضيح):

```
panic: runtime error: integer divide by zero
main.f(0)
    src/gopl.io/ch5/defer1/defer.go:14
main.f(1)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(2)
    src/gopl.io/ch5/defer1/defer.go:16
main.f(3)
    src/gopl.io/ch5/defer1/defer.go:16
main.main()
    src/gopl.io/ch5/defer1/defer .go:10
```

كما سنرى قريبًا، من الممكن أن تتعافى الوظيفة من الهلع دون أن تغلق البرنامج.

لأغراض التشخيص، تترك حزمة runtime المبرمج يتخلص من الرصة باستخدام نفس الآلية، أي تأجيل استدعاء printStack في الحزمة الرئيسية (main)،

```
gopl.io/ch5/defer2
func main() {
   defer printStack()
   f(3)
}
func printStack() {
   var buf [4096]byte
   n := runtime.Stack(buf[:], false)
   os.Stdout.Write(buf[:n])
}
```

يُطبع النص الإضافي التالي (المبسط للتوضيح) على الناتج القياسي:

```
goroutine 1 [running]:
main.printStack()
    src/gopl.io/ch5/defer2/defer.go:20
main.f(0)
    src/gopl.io/ch5/defer2/defer.go: 27
main.f(1)
    src/gopl.io/ch5/defer2/defer.go: 29
main.f(2)
    src/gopl.io/ch5/defer2/defer.go:29
main.f(3)
    src/gopl.io/ch5/defer2/defer.go: 29
main.main()
    src/gopl.io/ch5/defer2/defer.go:15
```

قد يتفاجأ القراء المطلعون على الاستثناءات في اللغات الأخرى من أن runtime.Stack يمكنها طباعة معلومات حول وظائف تم "تفكيكها" بالفعل. تشغيل آلية هلع Go الوظائف المؤجلة قبل أن تفكك الرصة.

5.10 الاسترجاع/التعافي - Recover

إن الاستسلام هو الاستجابة المناسبة للهلع عادة، ولكن ليس هذا هو الحال دائمًا. قد يكون التعافي واسترجاع البرنامج ممكنًا بطريقة ما، على الأقل لتنظيف الفوضى قبل إنهاء البرنامج. على سبيل المثال، خادم الويب الذي يواجه مشكلة غير متوقعة يمكن أن يغلق الاتصال بدلًا من ترك العميل معلقًا، وقد يقدم تقرير بالخطأ للعميل أيضًا أثناء التطوير.

لو استُدعيت وظيفة recover المدمجة داخل وظيفة مؤجلة، وحدث هلع بالوظيفة التي تحتوي على عبارة defer، فإن الاسترجاع ينهي حالة الهلع الحالية، ويعيد قيمة الهلع. لا تكمل الوظيفة التي حدث بها الهلع عملها من حيث توقفت، ولكنها تعود للعمل بشكل طبيعي. لو استُدعيت وظيفة recover في أي وقت آخر، لن يكون لها تأثير وستعيد nil.

للتوضيح، فكر في تطوير محلل للغة ما، حتى لو ظهر أنه يعمل بشكل جيد، سيظل هناك حالات خلل في أركان مظلمة غير واضحة، نظرًا لتعقيد وظيفة المحلل. قد نفضل أن يحول المحلل حالات الهلع هذه إلى أخطاء تحليل عادية بدلًا من انهيار البرنامج، ربما مع عرض رسالة إضافية تحث المستخدم على إرسال تقرير بالخلل.

```
func Parse(input string) (s *Syntax, err error) {
    defer func() {
        if p := recover(); p != nil {
            err = fmt.Errorf("internal error: %v", p)
        }
    }()
    // ... parser...
}
```

تتعافى الوظيفة المؤجلة في Parse من الهلع، وتستخدم قيمة الهلع لبناء رسالة خطأ، وقد تتضمن النسخة الأرقى رصة استدعاء كاملة باستخدام runtime.Stack. تُعيّن الوظيفة المؤجلة بعد ذلك إلى النتيجة err، والتي تُعاد للمستدعي.

إن التعافي دون تمييز من حالات الهلع هو إجراء مخادع لأن حالة متغيرات الحزمة بعد الهلع نادرًا ما تكون مُعرّفة أو موثقة جيدًا. قد يكون هناك تحديث ضروري لبنية البيانات غير مكتمل، أو يُترك اتصال بملف أو شبكة مفتوحًا دون أن يُغلق، أو يتم الحصول على قفل ولكن دون تحريره. علاوة على ذلك، الاسترجاع أو التعافي غير المميز، عن طريق استبدال الانهيار بسطر في ملف سجل مثلًا، يمكن أن يؤدي لمرور الخلل دون ملاحظة.

يمكن للتعافي من الهلع داخل نفس الحزمة أن يساعد على تبسيط التعامل مع الأخطاء المعقدة وغير المتوقعة، ولكن كقاعدة عامة، لا يجب أن تحاول التعافي من هلع حزمة أخرى. يجب عل APIs العامة أن تقدم حالات الفشل كأخطاء. بالمثل، لا يجب أن تتعافى من الهلع الذي يمكن أن يمر عبر وظيفة لا تحفظها، مثل معاودة المستدعي للاستدعاء، حيث أنك لا تستطيع التأكد من سلامته.

على سبيل المثال، تقدم حزمة net/http خادم ويب يوزع الطلبات القادمة على وظائف مُداوِل يقدمها المستخدِم. بدلًا من ترك الهلع في أحد هذه المُداوِلات يغلق العملية، يستدعي الخادم الوظيفة recover، ويطبع أثر رصة، ويستمر في الخدمة. هذا مريح أثناء الممارسة، ولكنه يخاطر بتسريب الموارد أو ترك مداول فاشل في حالة غير محددة يمكن أن تؤدي إلى مشكلات أخرى.

لكل الأسباب المذكورة أعلاه، سيكون من الآمن أن نسترجع/نتعافى بشكل انتقائي، هذا لو استرجعنا على الإطلاق. بصيغة أخرى، تعافى من حالات الهدف المُصممة للتعافي منها فقط، وهي نادرة. يمكن ترميز هذه النية من خلال استخدام نوع مميز وغير متوقع لقيمة الهلع، واختبار ما إذا كانت القيمة التي تعيدها الوظيفة recover تحمل نفس النوع. (سنرى طريقة لفعل هذا في المثال التالي). لو كان الأمر كذلك، سنقدم تقرير بالهلع كخطأ عادي، ولو لم يكن الوضع كذلك، نستدعي panic بنفس القيمة لمواصلة حالة الهلع.

إن المثال أدناه هو تنويع على برنامج title الذي يقدم تقرير بالخطأ لو كان مستند HTML يحتوي على عناصر <title> متعددة. لو كان الأمر كذلك، فإنه يحبط التكرار من خلال استدعاء panic بقيمة إنقاذ من نوع خاص.

```
gopl.io/ch5/title3
// soleTitle returns the text of the first non-empty title element
// in doc, and an error if there was not exactly one.
func soleTitle(doc *html.Node) (title string, err error) {
	type bailout struct{}
	defer func() {
		switch p := recover(); p {
		case nil:
			// no panic
		case bailout{}:
			// "expected" panic
			err = fmt.Errorf("multiple title elements")
		default:
			panic(p) // unexpected panic; carry on panicking
		}
	}()
	// Bail out of recursion if we find more than one non-empty title.
	forEachNode(doc, func(n *html.Node) {
		if n.Type == html.ElementNode && n.Data == "title" &&
			n.FirstChild != nil {
			if title != "" {
				panic(bailout{}) // multiple title elements
			}
			title = n.FirstChild.Data
		}
	}, nil)
	if title == "" {
		return "", fmt.Errorf("no title element")
	}
	return title, nil
}
```

تستدعي وظيفة المداول المؤجلة recover، وتفحص قيمة الهلع، وتقدم تقرير بخطأ عادي لو كانت القيمة {}bailout**.**

تشير كل القيم الأخرى التي ليست nil إلى هلع غير متوقع، وفي تلك الحالة يستدعى المداول panic بتلك القيمة، ويحل تأثير recover، ويتابع حالة الهلع الأصلية. (ينتهك هذا المثال نصيحتنا بشأن عدم استخدام الهلع في الأخطاء المتوقعة، ولكنه يقدم توضيح مختصر للآليات).

لا يوجد استرجاع أو تعافي من بعض الحالات، مثل نفاد الذاكرة، مما يجعل زمن تشغيل Go ينهي البرنامج بخطأ فادح.

تمرين 5.19: استخدم panic و recover لكتابة وظيفة لا تحتوي على عبارة إعادة ولكنها تعيد قيمة غير صفرية.

6- الطُرق - Methods

هيمنت البرمجة الكائنية (OOP) (object-oriented programming) منذ بداية التسعينات على نموذج البرمجة في الصناعة والتعليم، وقد تضمنت كل اللغات المستخدَمة على نطاق واسع منذ ذلك الحين دعمًا لها، ولغة Go ليست استثناء.

لا يوجد تعريف عام مقبول للبرمجة الكائنية، لذلك - ولأغراض هذا الكتاب - سنُعرّف الكائن "object" ببساطة بأنه قيمة أو متغير يحتوي على طُرق (methods)، والطريقة هي وظيفة مرتبطة بنوع محدد. ويُعرّف البرنامج الكائني بأنه البرنامج الذي يستخدم طرق للتعبير عن خصائص وعمليات كل هيكل بيانات حتى لا يحتاج العملاء إلى الدخول إلى تمثيل الكائن بشكل مباشر.

استخدمنا بانتظام في الفصول السابقة طرق من المكتبة القياسية، مثل "الثوان/Seconds" وهي أسلوب خاص بالنوع time.Duration:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

وعرّفنا طريقة خاصة بنا في القسم 2.5، وهي طريقة سلسلة النوع Celsius:

```
func (c Celsius) String() string { return fmt.Sprintf("%g°C", c) }
```

هذا الفصل هو أول من فصل من فصلين حول البرمجة الكائنية، وسنوضح فيه كيف يمكننا تعريف واستخدام الطُرق بكفاءة. سنغطي أيضًا مبدأين أساسيين في البرمجة الكائنية وهما التغليف (encapsulation) والتركيب (composition).

6.1 إعلانات الطريقة

تُعلن الطريقة بتنويعة من تنويعات إعلان الوظيفة العادية، ويظهر فيه مؤشر إضافي قبل اسم الوظيفة. يربط المؤشر الوظيفة مع نوع هذا المؤشر.

لنكتب أول طريقة في حزمة بسيطة خاصة بالهندسة المستوية:

```
gopl.io/ch6/geometry
package geometry
import "math"
type Point struct{ X, Y float64 }
// traditional function
func Distance(p, q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
// same thing, but as a method of the Point type
func (p Point) Distance(q Point) float64 {
    return math.Hypot(q.X-p.X, q.Y-p.Y)
}
```

يُطلق على المؤشر الإضافي p اسم مُستقبِل الطريقة (method's receiver)، وهو تراث من اللغات الكائنية المبكرة التي كانت تُسمي الطريقة "إرسال رسالة إلى كائن".

نحن لا نستخدم في لغة Go اسم مميز مثل this أو self للمستقبِل، بل نختار أسماء المستقبِل كما نختار اسم أي معامل آخر. ونظرًا لأن اسم المستقبِل سيُستخدم بشكل متكرر، سيكون من الأفضل اختيار اسم قصير ومتسق عبر الطرق المختلفة. والاختيار الشائع هو الحرف الأول من اسم النوع، مثل حرف p في Point.

يظهر معاملات المستقبِل في "استدعاء الطريقة" أمام اسم الطريقة، وهذا موازٍ للإعلان، وفيه يظهر مؤشر المستقبِل قبل اسم الطريقة.

```
p := Point{1, 2}
q := Point{4, 6}
fmt.Println(Distance(p, q)) // "5", function call
fmt.Println(p.Distance(q)) // "5", method call
```

لا يوجد تضارب بين إعلانيّ الوظيفة المسماة Distance أعلاه، فالأول يعلن عن وظيفة على مستوى الحزمة اسمها geometry.Distance، والثاني يعلن عن طريقة من النوع Point، لذا اسمها هو Point.Distance.

إن التعبير p.Distance يُطلَق عليه مُنتَقِي (Selector)، لأنه ينتقي طريقة Distance المناسبة للمستقبِل p من النوع Point. يُستخدم المنتقى كذلك في اختيار حقول البنية struct، كما هو الحال في p.X. وحيث أن الطرق والحقول تشغل نفس مساحة الاسم، فإن الإعلان عن الطريقة X في struct الخاص بـ Point سيكون غامضا، وسيرفضه المترجم.

يمتلك كل نوع مساحة خاصة لاسمه لأجل الطرق، ويمكننا استخدام الاسم Distance في طُرق أخرى طالما أنها تنتمي إلى أنواع مختلفة. لنُعرّف النوع Path الذي يمثل تسلسل من قطاعات الخط، ونمنحه طريقة Distance أيضًا.

```
// A Path is a journey connecting the points with straight lines.
type Path []Point
// Distance returns the distance traveled along the path.
func (path Path) Distance() float64 {
```

```
    sum := 0.0
    for i := range path {
        if i > 0 {
            sum += path[i-1].Distance(path[i])
        }
    }
    return sum
}
```

إن Path هو اسم نوع شريحة، وليس نوع Struct مثل Point، ولكن لازال بإمكاننا تعريف طرق لأجله. عند السماح للطرق بالارتباط بأي نوع، تختلف Go عن أي لغة كائنية أخرى، فعادة ما يكون من الملائم تعريف السلوكيات الإضافية لأنواع بسيطة مثل الأرقام أو السلاسل أو الشرائح أو الخرائط، أو أحيانًا الوظائف نفسها. ويمكن إعلان الطرق في أي نوع مُسمّى مُعرف في نفس الحزمة، طالما أن نوعها الضمني ليس مؤشر ولا واجهة.

تمتلك طريقتيّ Distance أنواع مختلفة، وهم ليسوا مرتبطين ببعضهما على الإطلاق، بالرغم من أن Path.Distance يستخدم Point.Distance داخليًا لحساب طول كل قطاع يربط بين النقاط المتجاورة.

لنستدعي الطريقة الجديدة لحساب محيط مثلث قائم الزاوية:

```
perim := Path{
    {1, 1},
    {5, 1},
    {5, 4},
    {1, 1},
}
fmt.Println(perim.Distance()) // "12"
```


في الاستدعاءين أعلاه للطرق المسماة Distance، حدد المترجم الوظيفة التي سيتم استدعاءها بناء على كل مِن اسم الطريقة ونوع المستقبِل. في الاستدعاء الأول، كان path[i-1] يحتوي على النوع Point، وبالتالي تم استدعاء Point.Distance، وفي الثانية كان perim يحتوي على النوع Path، وبالتالي تم استدعاء Path.Distance.

يجب أن تمتلك كل الطرق الخاصة بنوع معين أسماء متفردة، ولكن الأنواع المختلفة يمكنها استخدام نفس الاسم للطريقة، مثل أساليب Distance لـ Point و Path، ولا توجد حاجة لتخفيف أسماء الوظيفة (مثال: PathDistance) إلى اسم شديد الوضوح. سنرى هنا أول منفعة لاستخدام الطرق بدلًا من الوظائف العادية: أسماء الطرق يمكن أن تكون أقصر. وتزداد الفائدة في الاستدعاءات النابعة من خارج الحزمة، حيث أنه بإمكانها استخدام اسم أقصر وحذف اسم الحزمة أيضًا:

```
import "gopl.io/ch6/geometry"
```

```
perim := geometry.Path{{1, 1}, {5, 1}, {5, 4}, {1, 1}}
fmt.Println(geometry.PathDistance(perim)) // "12", standalone function
fmt.Println(perim.Distance())             // "12", method of geometry.Path
```

6.2 الطرق ذات مُستقبِل المؤشر

إن استدعاء وظيفة يصنع نسخة من قيمة كل مُعطى، ولو كانت الوظيفة بحاجة لتحديث المتغير، أو لو كان المعطى كبيرا جدًا لدرجة أننا نرغب في تجنب نسخه، فيجب أن نمرر عنوان المتغير باستخدام المؤشر. ينطبق نفس الأمر على الطرق التي تحتاج إلى تحديث متغير المستقبِل، حيث نربطها بنوع المؤشر مثل Point*.

```
func (p *Point) ScaleBy(factor float64) {
    p.X *= factor
    p.Y *= factor
}
```

إن اسم الطريقة هو (Point*).ScaleBy، والأقواس ضرورية، فبدونها سيُحلَل التعبير كـ (Point.ScaleBy)*.

إن التقليد في البرنامج الواقعي هو لو أن أي طريقة Point تمتلك مستقبِل مؤشر، فإن "كل" طرق Point يجب أن تمتلك مستقبِل مؤشر، حتى التي لا تحتاجه بالضرورة. لقد كسرنا هذه القاعدة لأجل Point حتى يمكننا أن نوضح كلا نوعيّ الطريقة.

إن الأنواع المسمّاة (Point) والمؤشرات الخاصة بها (Point*) هي الأنواع الوحيدة التي يمكن أن تظهر في إعلان المستقبِل. علاوة على ذلك، ولتجنب الغموض، لا يُسمح بإعلانات الطريقة في الأنواع المسماة التي تعتبر أنواع مؤشر في حد ذاتها:

```
type P *int
func (P) f() { /* ... */ } // compile error: invalid receiver type
```

يمكن استدعاء طريقة (Point*).ScaleBy من خلال توفير مستقبِل Point*، كالتالي:

```
r := &Point{1, 2}
r.ScaleBy(2)
fmt.Println(*r) // "{2, 4}"
```

أو كالتالي:

```
p := Point{1, 2}
pptr := &p
pptr.ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

أو كالتالي:

```
p := Point{1, 2}
(&p).ScaleBy(2)
fmt.Println(p) // "{2, 4}"
```

ولكن آخر حالتين فوضويتين. لحسن الحظ، تساعدنا اللغة هنا. لو كان المستقبِل p هو متغير من النوع Point ولكن الطريقة تتطلب مستقبل *Point، يمكننا استخدام هذا الاختصار:

```
p.ScaleBy(2)
```

وسيؤدي المترجم p& ضمني على المتغير. ينجح هذا فقط بالنسبة للمتغيرات، التي تشمل حقول struct مثل p.X، وعناصر المصفوفة أو الشريحة مثل perim[0]. لا يمكنني استدعاء طريقة *Point في مستقبل Point لا يمكن توجيهه، لأن لن تكون هناك طريقة للحصول على عنوان القيمة المؤقتة.

```
Point{1, 2}.ScaleBy(2) // compile error: can't take address of Point literal
```

لكن يمكننا استدعاء طريقة Point كـ Point.Distance مع مستقبِل *Point، لأن هناك طريقة للحصول على القيمة من العنوان وهي: قم بتحميل القيمة التي يشير لها المستقبِل وحسب. يُدخل المترجم عملية *operation ضمنية لأجلنا. واستدعائيّ الوظيفة هذين متكافئين:

```
pptr.Distance(q)
(*pptr).Distance(q)
```

لنلخص هذه الحالات الثلاثة مرة أخرى، حيث أنها تمثل نقطة ارتباك متكررة. في كل تعبير صحيح عن استدعاء الطريقة، تكون واحدة بالضبط من هذه العبارات الثلاثة صحيحة.

إما أن معطى المستقبل يمتلك نفس نوع مؤشر المستقبِل، كمثال، كلاهما يمتلك النوع T أو كلاهما يمتلك النوع *T:

```
Point{1, 2}.Distance(q) // Point
pptr.ScaleBy(2)         // *Point
```

أو أن معطى المستقبِل متغير من النوع T ومؤشر المستقبل من النوع *T. ويأخذ المترجم ضمنًا عنوان المتغير:

```
p.ScaleBy(2) // implicit (&p)
```

أو أن معطى المستقبل يمتلك النوع *T ومؤشر المستقبل يمتلك النوع T. ويزيل المترجم ضمنًا الإشارة للمستقبل، أو بمعنى آخر.. يحمّل القيمة:

```
pptr.Distance(q) // implicit (*pptr)
```

لو كانت كل الطرق الخاصة بالنوع المسمى T تمتلك نوع مستقبل لـ T نفسها (وليس T*)، فسيكون من الآمن نسخ أمثلة هذا النوع، واستدعاء أي من طرقه سيصنع نسخة بالضرورة. كمثال، قيم time.Duration تُنسخ بحرية، وهذا يشمل معطيات الوظائف. ولكن لو كانت الطريقة ذات مستقبل مؤشر، فيجب أن تتجنب أمثلة T لأن فعل هذا يمكن أن ينتهك الثوابت الداخلية. كمثال، نسخ مثال خاص بـ bytes.Buffer سيجعل الأصل والنسخة يسندان (انظر 2.3.2) إلى نفس المصفوفة الداخلية من البايتات. ستكون استدعاءات الطريقة التالية ذات آثار غير متوقعة.

6.2.1 Nil هي قيمة مستقبِل صحيحة

مثلما تسمح بعض الوظائف بمؤشرات Nil كمعطيات، تسمح بها بعض الطرق أيضًا مع مستقبلاتها، وخاصة لو كانت nil هي قيمة صفر مهمة في النوع، كما هو الحال في الخرائط والشرائح. تمثل nil قائمة فارغة في قائمة الأعداد الصحيحة البسيطة المرتبطة التالية:

```
// An IntList is a linked list of integers.
// A nil *IntList represents the empty list.
type IntList struct {
	Value int
	Tail *IntList
}
// Sum returns the sum of the list elements.
func (list *IntList) Sum() int {
	if list == nil {
		return 0
	}
	return list.Value + list.Tail.Sum()
}
```

عندما تُعرِّف نوعا تسمح طرقه بـ nil كقيمة مستقبل، فيجب الإشارة إلى هذا صراحة في التعليق التوثيقي كما فعلنا أعلاه.

إليك جزء من تعريف نوع Values من حزمة net/url:

```
net/url
package url
// Values maps a string key to a list of values.
type Values map[string][]string
// Get returns the first value associated with the given key,
// or "" if there are none.
func (v Values) Get(key string) string {
   if vs := v[key]; len(vs) > 0 {
      return vs[0]
      }
   return ""
```

```
}
// Add adds the value to key.
// It appends to any existing values associated with key.
func (v Values) Add(key, value string) {
    v[key] = append(v[key], value)
}
```

يعرض تمثيلها كخريطة ولكنه يقدم أيضًا طرق لتبسيط الوصول إلى الخريطة، والتي توجد قيمها في شكل شرائح أو سلاسل – إنها خريطة متعددة "multimap". يمكن لعملاءها استخدام المشغلين الداخليين الخاصين بها (make, slice literals, m[key] ، وغيرهم)، أو طرقها، أو كلاهما، حسبما يفضلون:

```
gopl.io/ch6/urlvalues
m := url.Values{"lang": {"en"}} // direct construction
m.Add("item", "1")
m.Add("item", "2")
fmt.Println(m.Get("lang")) // "en"
fmt.Println(m.Get("q"))    // ""
fmt.Println(m.Get("item")) // "1"      (first value)
fmt.Println(m["item"])     // "[1 2]"  (direct map access)
m = nil
fmt.Println(m.Get("item")) // ""
m.Add("item", "3")         // panic: assignment to entry in nil map
```

في الاستدعاء الأخير لـ Get، يتصرف مستقبِل nil كخريطة فارغة. ويمكننا أن نكتبه بالمثل كـ Values(nil).Get("item"))، ولكن nil.Get("item") لن يترجَم لأن نوع Nil لم يُحدد. وعلى النقيض، الاستدعاء الأخير لـ Add يصاب بالفوضى عند محاولته تحديث خريطة nil.

إن url.Values هي نوع خريطة، والخريطة تشير إلى ثنائيات المفتاح/القيمة الخاصة بها بشكل غير مباشر، وأي تحديثات أو حذف تقوم بها url.Values.Add على عناصر الخريطة تكون واضحة للمستدعي. مع ذلك، وكما هو الحال في الوظائف العادية، فإن أي تغييرات تقوم بها الطريقة على المرجع نفسه، مثل ضبطه على nil أو جعله يشير إلى هيكل بيانات خريطة مختلفة، لن تظهر للمستدعي.

6.3 تركيب الأنواع من خلال تضمين struct

فكر في النوع ColoredPoint:

```
gopl.io/ch6/coloredpoint
import "image/color"
type Point struct{ X, Y float64 }
type ColoredPoint struct {
```

```
    Point
    Color color.RGBA
}
```

يمكننا تعريف ColoredPoint كـ struct لثلاث حقول، ولكن بدلًا من ذلك نقوم بتضمين Point لتوفير حقول X و Y. وكما رأينا في القسم 4.4.3، فإن التضمين يدعنا نأخذ اختصار تركيبي لتعريف ColoredPoint الذي يحتوي على كل حقول Point، بالإضافة إلى غيرها. لو أردنا، يمكننا اختيار حقول ColoredPoint التي ساهم بها Point المتضمن بدون ذِكر Point:

```
var cp ColoredPoint
cp.X = 1
fmt.Println(cp.Point.X) // "1"
cp.Point.Y = 2
fmt.Println(cp.Y) // "2"
```

تُطبق آلية مشابهة على طُرق Point، ويمكننا أن نستدعي طرق حقل Point الضمني باستخدام مستقبِل من نوع ColoredPoint، رغم أن ColoredPoint ليس له أساليب معلنة:

```
red := color.RGBA{255, 0, 0, 255}
blue := color.RGBA{0, 0, 255, 255}
var p = ColoredPoint{Point{1, 1}, red}
var q = ColoredPoint{Point{5, 4}, blue}
fmt.Println(p.Distance(q.Point)) // "5"
p.ScaleBy(2)
q.ScaleBy(2)
fmt.Println(p.Distance(q.Point)) // "10"
```

إن طرق Point تم الترويج لها في ColoredPoint، وبهذه الطريقة، يسمح التضمين ببناء الأنواع المعقدة ذات الطرق المتعددة من خلال "تركيب/composition" حقول متعددة، يقدم كل منها طرق قليلة.

قد يشعر القراء المعتادين على اللغات الكائنية المعتمدة على الفئة بالرغبة في النظر إلى Point كفئة أساسية، و ColoredPoint كفئة فرعية أو فئة مشتقة، أو تفسير العلاقة بين هذه الأنواع كما لو أن ColoredPoint "هو" Point، ولكن سيكون هذا خاطئ. لاحظ استدعاءات Distance أعلاه. تمتلك Distance مؤشر من النوع Point، و q ليست Point، لذا بالرغم من أن q لها حقل مدمج من هذا النوع، إلا أننا يجب أن نختارها صراحة، ومحاولة تجاوز q ستكون خاطئة:

```
p.Distance(q) // compile error: cannot use q (ColoredPoint) as Point
```

إن ColoredPoint ليس Point، ولكنه "يمتلك" Point، ويحتوي على طريقتين إضافيتين هما Distance و ScaleBy تم ترويجهما من Point. لو كنت تفضل التفكير من حيث التطبيق، فإن الحقل المدمج يوجه المترجم لتقديم طرق تغليف إضافية تنوب عن الطرق المعلنة وتكافئ:

```
func (p ColoredPoint) Distance(q Point) float64 {
     return p.Point.Distance(q)
}
func (p *ColoredPoint) ScaleBy(factor float64) {
     p.Point.ScaleBy(factor)
}
```

عند استدعاء أولى طرق التغليف هذه لـ Point.Distance، فإن قيمة مستقبِله تكون p.Point وليس p، ولا يوجد وسيلة تُمكّن هذه الطريقة من دخول ColoredPoint المُدمج فيها Point.

إن نوع الحقل المجهول قد يكون مؤشر "pointer" لنوع مُسمّى، وفي هذه الحالة يتم الترويج للحقول والأساليب بشكل غير مباشر عبر الكائن المُشار إليه. إن إضافة مستوى آخر من عدم المباشرة يجعلنا نشارك الهياكل الشائعة، ويحقق تفاوت في العلاقات بين الكائنات ديناميكيًا. إن إعلان ColoredPoint أدناه يتضمن Point*:

```
type ColoredPoint struct {
     *Point
     Color color.RGBA
}
p := ColoredPoint{&Point{1, 1}, red}
q := ColoredPoint{&Point{5, 4}, blue}
fmt.Println(p.Distance(*q.Point)) // "5"
q.Point = p.Point    // p and q now share the same Point
p.ScaleBy(2)
fmt.Println(*p.Point, *q.Point) // "{2 2} {2 2}"
```

قد يحتوي نوع struct على أكثر من حقل مجهول واحد، لو أعلنا ColoredPoint على أنه:

```
type ColoredPoint struct {
     Point
     color.RGBA
}
```

فإن قيمة هذا النوع ستحتوي على كل طُرق Point، وكل طرق RGBA، وأي طرق إضافية مُعلنة في ColoredPoint بشكل مباشر. وعندما يقوم مترجم بتحويل مُنتقي مثل p.ScaleBy إلى طريقة، فإنه يبحث في البداية عن طريقة مُعلنة مباشرة باسم ScaleBy، ثم عن طرق رُوجت لمرة واحدة من حقول ColoredPoint المدمجة، ثم عن طرق رُوجت مرتين من حقول مدمجة داخل Point و RGBA وغيرها. يقدم المترجم تقرير خطأ لو كان المنتقي غامض أو مبهم ويحدث هذا لو قدّم المنتقي طريقتين تم ترويجهما من نفس الرتبة.

يمكن إعلان الطُرق فقط في الأنواع المسماة (مثل Point)، والمؤشرات التي تشير لها (Point*)، ولكن بفضل التضمين، أصبح من الممكن بل ومن المفيد أحيانًا أن تمتلك أنواع struct غير المسماة طُرق أيضًا.

إليك خدعة لطيفة لتوضيح الأمر. يوضح هذا المثال جزء من مخبأ (cache) طُبق باستخدام متغيرين على مستوى الحزمة وهما mutex (انظر 9.2) والخريطة التي يحرسها:

```
var (
	mu sync.Mutex // guards mapping
	mapping = make(map[string]string)
)
func Lookup(key string) string {
	mu.Lock()
	v := mapping[key]
	mu.Unlock()
	return v
}
```

إن النسخة أدناه مكافِئة لها وظيفيًا ولكنها تجمع معا المتغيرين المرتبطين في متغير واحد على مستوى الحزمة، مخبأ:

```
var cache = struct {
	sync.Mutex
	mapping map[string]string
}{
	mapping: make(map[string]string),
}
func Lookup(key string) string {
	cache.Lock()
	v := cache.mapping[key]
	cache.Unlock()
	return v
}
```

يمنح المتغير الجديد أسماء معبرة أكثر للمتغيرات المرتبطة بالمخبأ، ولأن حقل sync.Mutex مدمج فيها، فإن طرق قفله وفتحه تُروّج إلى نوع struct غير مُسمّى، مما يسمح لنا بإغلاق المخبأ باستخدام تركيب يشرح نفسه بنفسه.

6.4 قيم وتعبيرات الطريقة

نحن عادة ما نختار ونستدعي طريقة في نفس التعبير، كما في p. Distance()، ولكن من الممكن الفصل بين العمليتين. ينتج عن المُنتَقِي p.Distance "قيمة الطريقة" (method value)، وهي وظيفة تربط الطريقة (Point.Distance) مع قيمة مستقبل محددة p. ويمكن استثارة تلك الوظيفة بعد ذلك دون قيمة المستقبل، وهي تحتاج فقط إلى معطيات غير متعلقة بالمستقبِل.

```
p := Point{1, 2}
q := Point{4, 6}
distanceFromP := p.Distance                  // method value
```

```
fmt.Println(distanceFromP(q))        //"5"
var origin Point                          // {0, 0}
fmt.Println(distanceFromP(origin))   // "2.23606797749979", √5

scaleP := p.ScaleBy                  // method value
scaleP(2)                            // p becomes (2, 4)
scaleP(3)                            // then (6, 12)
scaleP(10)                                // then (60, 120)
```

إن قيم الطريقة مفيدة عندما تستدعي API الحزمة قيمة الوظيفة، وعندما يكون سلوك العميل المطلوب في هذه الوظيفة هو استدعاء الطريقة لمستقبِل معين. كمثال، تستدعي الوظيفة time.AfterFunc قيمة وظيفة بعد تأخير محدد. ويستخدمها هذا البرنامج لإطلاق الصاروخ r بعد 10 ثوان:

```
type Rocket struct { /* ... */ }
func (r *Rocket) Launch() {/*...*/}

r := new(Rocket)
time.AfterFunc(10 * time.Second, func() { r.Launch() })
```

تركيبة قيمة الطريقة أقصر:

```
time.AfterFunc(10 * time.Second, r.Launch)
```

يرتبط بقيمة الطريقة "تعبير الطريقة" (method expression). عند استدعاء الطريقة، كنقيض للوظيفة العادية، يجب أن نقدم المستقبِل بطريقة خاصة باستخدام تركيبة المنتقي. إن تعبير الطريقة – الذي يُكتب T.f أو (*T).f هو نوع، وينتج عنه قيمة وظيفة ذات مؤشر أول منتظم يأخذ مكان المستقبل، وبالتالي يمكن استدعاءه بالطريقة المعتادة.

```
p := Point{1, 2}
q := Point{4, 6}

distance := Point.Distance           // method expression
fmt.Println(distance(p, q))          // "5"
fmt.Printf("%T\n", distance)    // "func(Point, Point) float64"

scale := (*Point).ScaleBy
scale(&p, 2)
fmt.Println(p)                  // "{2 4}"
fmt.Printf("%T\n", scale)       // "func(*Point, float64)"
```

يمكن لتعبيرات الطريقة أن تكون مفيدة عندما تحتاج قيمة تمثل اختيار بين طرق متعددة تنتمي إلى نفس النوع، حتى يمكنك استدعاء الطريقة المختارة عن طريق مستقبلات عديدة مختلفة. يمثل المتغير op في المثال التالي إما طريقة جمع أو طرح في النوع Point، ويستدعيه Path.TranslateBy في كل نقطة على المسار:

```
type Point struct{ X, Y float64 }
func (p Point) Add(q Point) Point { return Point{p.X + q.X, p.Y + q.Y} }
func (p Point) Sub(q Point) Point { return Point{p.X - q.X, p.Y - q.Y} }
type Path []Point
func (path Path) TranslateBy(offset Point, add bool) {
     var op func(p, q Point) Point
     if add {
          op = Point.Add
     } else {
          op = Point.Sub
     }
     for i := range path {
          // Call either path[i].Add(offset) or path[i].Sub(offset).
          path[i] = op(path[i], offset)
}
}
```

6.5: مثال: نوع مُتَجه بِت (Bit VectorType)

تُطبق المجموعات في لغة Go عادة كـ map[T]bool، حيث T هي نوع العنصر. إن المجموعة التي تمثلها خريطة تكون مرنة جدًا، ولكن في مشكلات معينة، قد يتفوق التمثيل المتخصص عليها. كمثال، في نطاقات مثل تحليل تدفق البيانات، حيث عناصر المجموعة هي أعداد صحيحة صغيرة غير سالبة، تحتوي المجموعات على العديد من العناصر، وتشيع عمليات المجموعة مثل الاتحاد والتقاطع، وبالتالي يكون مُتجه بِت (bit vector) مثالي فيها.

يستخدم متجه بِت شريحة من قيم عدد صحيح غير مُوقّع أو "كلمات"، ويمثل كل بِت فيها عنصر محتمل في المجموعة. تحتوي المجموعة على i لو كان بِت i-th مُعَد. يوضح البرنامج التالي نوع متجه بِت بسيط بثلاث طرق:

```
gopl.io/ch6/intset
// An IntSet is a set of small non-negative integers.
// Its zero value represents the empty set.
type IntSet struct {
   words []uint64
}
// Has reports whether the set contains the non-negative value x.
func (s *IntSet) Has(x int) bool {
   word, bit := x/64, uint(x%64)
   return word < len(s.words) && s.words[word]&(1<<bit) != 0
}
// Add adds the non-negative value x to the set.
func (s *IntSet) Add(x int) {
   word, bit := x/64, uint(x%64)
   for word >= len(s.words) {
      s.words = append(s.words, 0)
   }
   s.words[word] |= 1 << bit
}
// UnionWith sets s to the union of s and t.
```

```
func (s *IntSet) UnionWith(t *IntSet) {
    for i, tword := range t.words {
        if i < len(s.words) {
            s.words[i] |= tword
        } else {
            s.words = append(s.words, tword)
        }
    }
}
```

حيث أن كل كلمة تحتوي على 64 بِت، سنستخدم القسم x/64 كمؤشر للكلمة، والباقي x%64 كمؤشر بِت داخل تلك الكلمة، حتى نتمكن من تحديد البِت الخاص بـ x فيها. تستخدم عملية UnionWith عامل OR البِتّي | لحساب اتحاد 64 عنصر في المرة الواحدة. (سنرجع إلى اختيار كلمات 64 بِت في التمرين 6.5).

ينقُص هذا التطبيق العديد من الخصائص المرغوبة، بعضها موجود كتمرينات أدناه، ولكن هناك خاصية من الصعب العمل بدونها وهي طريقة طباعة IntSet كسلسلة. لنرى طريقة String كما فعلنا في Celsius في القسم 2.5:

```
// String returns the set as a string of the form "{1 2 3}".
func (s *IntSet) String() string {
    var buf bytes.Buffer
    buf.WriteByte('{')
    for i, word := range s.words {
        if word == 0 {
            continue
        }
        for j := 0; j < 64; j++ {
            if word&(1<<uint(j)) != 0 {
                if buf.Len() > len("{") {
                    buf.WriteByte(' ')
                }
                fmt.Fprintf(&buf, "%d", 64*i+j)
            }
        }
    }
    buf.WriteByte('}')
    return buf.String()
}
```

لاحظ تشابه طريقة String أعلاه مع intsToString في القسم 3.5.4، ويُستخدم bytes.Buffer بهذه الطريقة عادة في طُرق String. تعامل حزمة fmt الأنواع بطريقة String خاصة حتى يمكن عرض قيم الأنواع المعقدة نفسها بطريقة سهلة على المُستخدِم. بدلًا من طباعة التمثيل الخام للقيمة (وهو struct في هذه الحالة)، تستدعي fmt طريقة String. تعتمد الآلية على الواجهات وتأكيدات النوع، وهو ما سنشرحه في الفصل السابع.

يمكننا الآن توضيح IntSet بشكل عملي:

```
var x, y IntSet
x.Add(1)
x.Add(144)
```

```
x.Add(9)
fmt.Println(x.String()) // "{1 9 144}"

y.Add(9)
y.Add(42)
fmt.Println(y.String()) // "{9 42}"

x.UnionWith(&y)
fmt.Println(x.String()) // "{1 9 42 144}"
fmt.Println(x.Has(9),x.Has(123)) // "true false"
```

تنبيه: لقد أعلنا عن String و Has كطُرق لنوع المؤشر IntSet*، ليس لأن هذا ضروري، ولكن لأنها تتسق مع الطريقتين الأخريتين، اللتان تحتاجان إلى مستقبل مؤشر لأنهم خاصين بـ s.words. من ثمّ، لا تمتلك قيمة IntSet أسلوب String، وهو ما يؤدي من حين لآخر إلى مفاجآت مثل:

```
fmt.Println(&x)            // "{1 9 42 144}"
fmt.Println(x.String())    // "{1 9 42 144}"
fmt.Println(x)             // "{[4398046511618 0 65536]}"
```

في الحالة الأولى، نطبع مؤشر IntSet* الذي له بالفعل طريقة String. وفي الحالة الثانية، نستدعي String() في متغير IntSet، ويُدخل المترجم التضمين والعملية، ويمنحنا مؤشر يحتوي على طريقة String. ولكن في الحالة الثالثة، ولأن قيمة IntSet لا تحتوي على طريقة String، يطبع fmt.Println تمثيل struct بدلًا من ذلك. من المهم ألا تنسى عامل &. إن صنع طريقة String من IntSet بدلًا من IntSet* قد يكون فكرة جيدة، ولكن هذا يختلف من حالة إلى أخرى، ويخضع للحكم الشخصي على كل حالة.

تمرين 6.1: طبّق هذه الطُرق الإضافية.

```
func (*IntSet)  Len() int         // return the number of elements
func (*IntSet)  Remove(x int)     // remove x from the set
func (*IntSet)  Clear()           // remove all elements from the set
func (*IntSet)  Copy() *IntSet    // return a copy of the set
```

تمرين 6.2: عرّف طريقة (*IntSet).AddAll(... int) متباينة تسمح بإضافة قائمة بالقيم، مثل s.AddAll(1, 2, 3).

تمرين 6.3: يحسب (*IntSet).UnionWith اتحاد مجموعتين باستخدام |، وعامل OR بِتي موازي للكلمة. طبق الطرق الخاصة بـ IntersectWith و DifferenceWith و Sym-metricDifference لعمليات المجموعة المناظرة. (الاختلاف التماثلي بين المجموعتين التين تحتويان على عناصر موجودة في مجموعة أو الأخرى ولكن ليس في كلاهما).

تمرين 6.4: أضِف طريقة Elems التي تقدم شريحة تحتوي على عناصر من المجموعة مناسبة للتكرار عبر حلقة نطاق.

تمرين 6.5: نوع كل كلمة استخدمها IntSet هو uint64، ولكن حساب 64-bit قد يكون غير كافي على منصة 32-bit. عدّل البرنامج ليستخدم النوع uint، وهو نوع العدد الصحيح غير الموقّع الأكثر كفاءة للمنصة. بدلًا من القسمة على 64، عرّف "ثابت/constant" يحمل الحجم الفعال لـ uint في بِتات، 32 أو 64. ويمـكنك استخدام تعبير بارع مثل (63 << (0)uint^) << 32 لهذا الغرض:

6.6 التغليف Encapsulation

يُقال أن متغير أو طريقة الكائن "مغلفة" لو كان لا يمكن للعملاء الوصول إلى الكائن يُطلق على التغليف أحيانًا "تخبئة المعلومات"، وهو جانب أساسي في البرمجة الكائنية.

تمتلك Go آلية واحدة فقط للتحكم في ظهور الأسماء وهي أن المُعرّفات ذات الحروف الكبيرة تُصدَّر من الحزمة التي تُعرّف فيها، بينما أن الأسماء التي لا تُكتب بحروف كبيرة لا تُصدّر. إن نفس الآلية التي تحُد من الوصول لأعضاء الحزمة تحُد أيضًا الوصول إلى حقول struct أو طرق النوع. ونتيجة لذلك، يجب أن نجعل الكائن struct حتى نستطيع تغليفه.

هذا هو السبب في الإعلان عن نوع IntSet في القسم السابق كنوع struct بالرغم من أنه يمتلك حقل واحد فقط:

```
type IntSet struct {
    words []uint64
}
```

يمكننا بدلًا من ذلك تعريف IntSet كنوع شريحة كالتالي، ولكن بالطبع يجب أن نستبدل كل s.words ب s* في طُرُقه:

```
type IntSet []uint64
```

بالرغم من أن هذه النسخة من IntSet ستكون مكافئة للنسخة السابقة بشكل جوهري، إلا أنها ستسمح للعملاء من الحِزم الأخرى بقراءة وتعديل الشريحة مباشرة. بمعنى آخر، بينما يمكن استخدام التعبير s* في أي حزمة، إلا أن s.words يمكن أن تظهر فقط في الحزم التي تُعرِّف IntSet.

أحد النتائج الأخرى لهذه الآلية القائمة على النوع هي أن وحدة التغليف هي الحزمة وليس النوع كما هو الحال في العديد من اللغات الأخرى. إن حقول نوع Struct ظاهرة في كل الشفرة الموجودة داخل نفس الحزمة، ولا يهم هل ظهرت الشفرة في الوظيفة أم في الطريقة.

يقدم التغليف ثلاث فوائد. الفائدة الأولى، نظرًا لكون العملاء لا يستطيعون تعديل متغيرات الكائن بشكل مباشر، يحتاج الفرد لفحص عبارات أقل لفهم القيم المحتملة لهذه المتغيرات.

الفائدة الثانية، أن إخفاء تفاصيل التطبيق يمنع العملاء من الاعتماد على الأشياء التي يمكن أن تتغير، وهو ما يمنح المصمم حرية أكبر في تطوير التطبيق بدون كسر توافق API.

كمثال على ما ذكر، انظر النوع bytes.Buffer. يُستخدم هذا النوع بشكل متكرر لتجميع سلاسل قصيرة جدًا، من ثمّ، الاحتفاظ بمساحة إضافية صغيرة في الكائن لتجنب تخصيص الذاكرة في هذه الحالة يُعتبر من التحسينات المربحة. وحيث أن Buffer هو نوع struct، فإن هذه المساحة تأخذ شكل حقل إضافي في النوع [64]byte بالاسم ذي حروف صغيرة. عند إضافة هذا الحقل، ولأنه لا يُصدّر، لا ينتبه عملاء Buufer خارج حزمة bytes لأي تغيير إلا الأداء المُحسّن. إن Buffer وطريقة Grow الخاصة به موضحين أدناه، ومُقدمّان بشكل مبسط لأجل المزيد من التوضيح:

```
type Buffer struct {
     buf []byte
     initial [64]byte
     /* ...*/
}
// Grow expands the buffer's capacity, if necessary,
// to guarantee space for another n bytes. [...]
func (b *Buffer) Grow(n int) {
     if b.buf == nil {
          b.buf = b.initial[:0] // use preallocated space initially
     }
     if len(b.buf)+n > cap(b.buf) {
          buf := make([]byte, b.Len(), 2*cap(b.buf) + n)
     copy(buf, b.buf)
     b.buf = buf
     }
}
```

إن الفائدة الثالثة للتغليف، وأكثر الفوائد أهمية في العديد من الحالات، هي أنه يمنع العملاء من اختيار متغيرات كائن بشكل عشوائي. ونظرًا لأن متغيرات الكائن يمكن ضبطها فقط من خلال الوظائف الموجودة في نفس الحزمة، يمكن لكاتب هذه الحزمة أن يضمن حفاظ كل هذه الوظائف على الثوابت الداخلية للكائن. على سبيل المثال، يسمح نوع Counter أدناه للعملاء بزيادة العدّاد أو تصفيره، ولكن لا يسمح لهم بضبطه على قيمة عشوائية ما:

```
type Counter struct { n int }
func (c *Counter) N() int        { return c.n }
func (c *Counter) Increment()    { c.n++ }
func (c *Counter) Reset()        { c.n = 0 }
```

إن الوظائف التي تدخل أو تعدل القيم الداخلية للنوع وحسب، مثل طُرق نوع Logger من حزمة log أدناه، يُطلق عليها getters و setters. مع ذلك، فعند تسمية طريقة getter، عادة ما نحذف السابقة Get. يمتد هذا التفضيل للاختصار

ليشمل كل الطرف وليس فقط مَن يمكنهم دخول الحقل، ويشمل السوابق الأخرى الزائدة أيضًا مثل Fetch و Find و Lookup.

```
package log
type Logger struct {
    flags int
    prefix string
    // ...
}
func (l *Logger) Flags() int
func (l *Logger) SetFlags(flag int)
func (l *Logger) Prefix() string
func (l *Logger) SetPrefix(prefix string)
```

لا يمنع نمط Go تصدير الحقول، ولكن بمجرد تصديرها، لا يمكن للحقل إلغاء التصدير بدون إجراء تغيير غير متوافق على API، وبالتالي يجب أن تتمهل في الاختيار الأولي، وتدريس تعقيد الثوابت التي يجب الحفاظ عليها، والتغييرات المستقبلية المحتملة، وكمية شفرة العميل التي ستتأثر بالتغيير.

إن التغليف ليس مرغوب فيه دائمًا، وعند الكشف عن تمثيله كرقم int64 من النانو ثانية، فإن time.Duration يدعنا نستخدم كل العمليات الحسابية وعمليات المقارنة المعتادة في المُدد، بل وحتى لتعريف ثوابت هذا النوع:

```
const day = 24 * time.Hour
fmt.Println(day.Seconds()) // "86400"
```

المثال الآخر هو مقارنة IntSet مع نوع geometry.Path في بداية هذا الفصل. عُرِّف Path كنوع شريحة، وسمح لعملاءه ببناء أمثلة باستخدام التركيب الحرفي للشريحة، وتكرارها عبر نقاطها باستخدام حلقة نطاق، إلخ، بينما كانت هذه العمليات محظورة على عملاء IntSet.

إليك الفارق الجوهري بينهما: geometry.Path بطبيعته هو سلسلة من النقاط، لا أكثر لا أقل، ونحن لا نتوقع إضافة حقول جديدة له، لذا من المنطقي أن تكشف حزمة الهندسة عن أن Path هو شريحة. على النقيض، يُمثَل IntSet وحسب كشريحة uint64[]. ويمكن تمثيله باستخدام uint[]، أو بشيء مختلف جدًا بالنسبة للمجموعات القليلة أو الصغيرة جدًا، وقد يستفيد من الخصائص الإضافية مثل الحقل الإضافي لتسجيل عدد العناصر في المجموعة. ولهذه الأسباب، من المنطقي أن يكون IntSet معتم.

تعلمنا في هذا الفصل كيف نربط الطرق مع الأنواع المسماة، وكيف نستدعي تلك الطرق. وبالرغم من أن الطرق ضرورية في البرمجة الكائنية، إلا أنها لا تمثل سوى نصف الصورة فقط، ولإكمال الصورة، سنحتاج إلى الواجهات "interfaces" وهذا هو موضوع الفصل التالي.

7- الواجهات

تُعبّر الواجهات (Interface) بأنواعها عن العمومية أو الأفكار السطحية لسلوكيات الأنواع الأخرى، والفائدة من هذه العمومية هي تمكيننا من كتابة وظائف أكثر مرونة وتكيفًا لأن تلك الوظائف ليست مقيدة في تفاصيل تطبيق واحد فقط.

لدى معظم اللغات كائنية التوجه فكرة عامة عن الواجهات، ولكن ما يجعل واجهات لغة جو مميزة جدًا هو أنها مفهومة ضمنيًا، أي بمعنى آخر لا يوجد هناك الحاجة لإعلان جميع الواجهات التي تلبي نوع معين محدد، وبالتالي حيازة الطرق الضرورية كافي. يتيح لك هذا التصميم إمكانية إنشاء واجهات جديدة تتم تلبيتها بأنواع محددة موجودة دون تغيير الأنواع الموجودة، وبالتالي تكمن الفائدة في الأنواع المعرفة ضمن الحزم والتي لا يمكنك التحكم فيها.

في هذا الفصل سنتعلم عن أساسيات مبدأ عمل أنواع الواجهات وقيمهم، وسنتعلم عن العديد من الواجهات المهمة من المكتبة القياسية ، فالعديد من برامج جو تستخدم الواجهات القياسية بقدر استخدامها للواجهات الخاصة بها. وأخيرًا سنتعلم عن النوع التوكيدي (type assertions) (راجع القسم 7.10) والنوع التبديلي أو التحويلي (type switches) (راجع القسم 7.13) وعن دورهما في إتاحة مختلف أنواع العمومية.

7.1 الواجهات كعقود

(للتوضيح، المقصود بالعقود هو اتفاق ضمني أنت تعقده بين مستخدمي ومنفذي الواجهة، أي العقد يمثل كيفية استخدام الواجهة)

جميع الأنواع التي درسناها حتى الآن هي أنواع محددة (concrete types)، فالأنواع المحددة تحدد التمثيل الفعلي لقيمها وتبين العمليات الجوهرية لذلك التمثيل، مثل العمليات الحسابية على الأرقام أو مثل فهرسة وإلحاق وتحديد مدى الشرائح (slices)، وقد يظهر النوع المحدد سلوكيات إضافية عبر طرقه، فعند معرفتك لقيمة نوع محدد فإنك تعرفها بالضبط وتعرف ماذا تفعل بها.

يوجد نوع آخر من الأنواع في لغة جو يسمى نوع الواجهة (interface type)، فالواجهة هي عباره عن نوع مجرد (abstract type)، فهو لا يكشف البناء الداخلي أو تمثيل قيمته أو مجموعة العمليات الأساسية التي يدعمها، فهو يكشف

فقط بعض طرقه. فعندما تعرف قيمة نوع واجهة فإنك لا تعرف شيئًا عن ماهيته، فإنك تعرف فقط ما يمكنك فعله به أو بالأصح تعرف السلوكيات المقدمة ضمن طرقه.

أستخدمنا خلال هذا الكتاب دالتين متشابهتين لتنسيق السلسلة (string): الأولى هي fmt.Printf والتي بدورها تكتب النتائج على مخرج قياسي (أي ملف) والثانية fmt.Sprintf وتعيد النتائج كسلسلة نصية. سيكون من المؤسف إذا تكررت النتائج بسبب تلك الاختلافات السطحية في كيفية استخدام النتائج، ولكن بفضل الواجهات لا تتكرر النتائج، فإن كلا الدالتين عبارة عن غلاف لدالة ثالثة fmt.Fprintf لا تدري عما يحصل للنتائج التي تحسبها:

```
package fmt
func Fprintf(w io.Writer, format string, args ...interface{}) (int, error)
func Printf(format string, args ...interface{}) (int, error) {
    return Fprintf(os.Stdout, format, args...)
}
func Sprintf(format string, args ...interface{}) string {
    var buf bytes.Buffer
    Fprintf(&buf, format, args...)
    return buf.String()
}
```

حرف F الموجود بداية الأمر Fprintf يدل على كلمة ملف file ويشير إلى أن المخرج المنسق يجب أن يُكتب للملف المقدم كأول معطى، في حالة Printf إن المعطى os.Stdout, عبارة عن os.File*، وفي حالة Sprintf إن المعطى ليس ملفا وإنما شيء يشبه إلى حد ما فإن buf& مجرد مؤشر point er لمخزن الذاكرة المؤقة buffer لأي بايت يمكن كتابته.

والمعامل الأول لـ Fprintf ليس بملف أيضًا، إنه مجرد io.Writer وهو نوع واجهة بالإعلان التالي:

```
package io
// Writer is the interface that wraps the basic Write method.
type Writer interface {
    // Write writes len(p) bytes from p to the underlying data stream.
    // It returns the number of bytes written from p (0 <= n <= len(p))
    // and any error encountered that caused the write to stop early.
    // Write must return a non-nil error if it returns n < len(p).
    // Write must not modify the slice data, even temporarily.
    //
    // Implementations must not retain p.
    Write(p []byte) (n int, err error)
}
```

تُحدد الواجهة io.Writer العقد بين Fprintf ومستدعييها، من ناحية يتطلب العقد من المستدعي تقديم قيمة نوع العقد مثل os.File* أو bytes.Buffer* ولديه طريقة تسمى Write تشمل توقيع وسلوك لائق، ومن ناحية أخرى يضمن العقد أن Fprintf سيقوم بمهامه بأي قيمه ترضي واجهة io.Writer، قد لا يفترض Fprintf أنه يكتب على ملف أو على ذاكرة وإنما قدرته على استدعاء Write فقط.

لا تفترض fmt.Fprintf أي شيء بشأن تمثيل القيمة وتعتمد فقط على السلوكيات التي يضمنها عقد io.Writer وبالتالي يمكننا تمرير أي نوع محدد يلبي متطلبات io.Writer كأول جملة إلى fmt.Fprintf، تسمى حرية استبدال نوع بنوع آخر يلبي نفس الواجهة بالاستبدالية (substitutability)، فالاستبدالية تعتبر سمة أساسية للغات كائنية التوجه.

دعنا نختبر هذه الميزة باستخدام نوع جديد، تقوم طريقة Write للنوع ByteCounter* بكل بساطة بإحصاء البايتات المكتوبة لها قبل نبذها، (يجب التحويل لجعل النوعين len(p) و c* يتوافقان مع جملة التعيين =+)

```
gopl.io/ch7/bytecounter
type ByteCounter int
func (c *ByteCounter) Write(p []byte) (int, error) {
    *c += ByteCounter(len(p)) // convert int to ByteCounter
    return len(p), nil
}
```

وبما أن ByteCounter* تلبي عقد io.Writer فبإمكاننا تمريرها إلى Fprintf والتي تنسق سلسلتها لتتناسب هذا التغير، وبالتالي يقوم ByteCounter بجمع طول النتائج.

```
var c ByteCounter
c.Write([]byte("hello"))
fmt.Println(c) // "5", = len("hello")
c = 0 // reset the counter
var name = "Dolly"
fmt.Fprintf(&c, "hello, %s", name)
fmt.Println(c) // "12", = len("hello, Dolly")
```

هناك واجهة أخرى غير io.Write لها أهمية كبيرة في حزمة fmt، توفر Fprintf و Fprintln سبيلًا للأنواع لتتحكم بكيفية طباعة قيمهم. بالرجوع إلى القسم 2.5 ستجد أننا قمنا بتعريف طريقة String لنوع درجة الحرارة Celsius وذلك لطباعة الحرارة كـ"100°C"، وفي القسم 6.5 قمنا بتزويد IntSet* بطريقة String لتصيير المجموعات باستخدام ترميز المجموعة التقليدي مثل {1 2 3}. إعلان طريقة String تجعل النوع يلبي واحده من أكثر الواجهات استخدامًا ألا وهو fmt.Stringer :

```
package fmt
// The String method is used to print values passed
// as an operand to any format that accepts a string
// or to an unformatted printer such as Print.
type Stringer interface {
    String() string
}
```

سنوضح في القسم 7.10 كيفية استكشاف حزمة fmt للقيم التي تلبي هذه الواجهة.

تمرين 7.1: بالاستفادة من الأفكار من ByteCounter، قم بتطبيق عدادات (counters) للكلمات وللأسطر، استفد من الأمر bufio.ScanWords.

تمرين 7.2: اكتب الدالة CountingWriter وتوقيعها أدناه، يستقبل io.Writer ويرجع Writer جديدا يلتف حول الأصلي، ومؤشرا يشير إلى متغير int64 والذي قد يحتوي في أي لحظة على عدد البايت المكتوبة لـ Writer الجديد.

```
func CountingWriter(w io.Writer) (io.Writer, *int64)
```

تمرين 7.3: اكتب طريقة String للنوع tree* في gopl.io/ch4/treesort (راجع القسم 4.4) بحيث يكشف عن ترتيب القيم في النظام الشجري.

7.2 أنواع الواجهات

يحدد نوع الواجهة مجموعة من الطرق التي يجب أن يمتلكها النوع المحدد ليُعتبر حالة من تلك الواجهة.

يعتبر النوع io.Writer واحد من أكثر الواجهات استخدامًا لأنه يوفر تجريد لجميع الأنواع التي يمكن كتابة البايتات إليها، ويشمل ذلك الملفات ومخزن الذاكرة المؤقت واتصالات الشبكة وعملاء بروتوكول HTTP والأرشيفي والملبدين وغيرهم. تحدد الحزمة io العديد من الواجهات المفيدة الأخرى، تمثل واجهة Reader أي نوع يمكنك من خلاله قراءة البايتات، وتعتبر واجهة Closer أي قيمة يمكنك إغلاقها مثل ملف أو اتصال بالشبكة. (حاليا ربما قد لاحظت اصطلاح التسمية للعديد من واجهات الطريقة الوحيدة (single-method interfaces) للغة جو).

```
package io
type Reader interface {
    Read(p []byte) (n int, err error)
}
type Closer interface {
    Close() error
}
```

ولقد وجدنا المزيد من إعلانات أنواع الواجهات الجديدة كخليط بين أنواع موجودة مسبقًا، ها هنا مثالان:

```
type ReadWriter interface {
    Reader
    Writer
}
type ReadWriteCloser interface {
    Reader
    Writer
    Closer
```

```
}
```

إن الصياغة (بناء الجملة) الموضحة أعلاه التي تشبه بنية التضمين تمكننا من تسمية واجهة أخرى كاختصار لكتابة جميع طرقها، فتسمى هذه الطريقة تضمين الواجهات (embedding)، ومن الممكن كتابة io.ReadWriter بدون تضمين ولكن ستكون الجملة طويلة كالتالي:

```
type ReadWriter interface {
     Read(p []byte) (n int, err error)
     Write(p []byte) (n int, err error)
}
```

ويمكن استخدام مزيج بين الإسلوبين:

```
type ReadWriter interface {
     Read(p []byte) (n int, err error)
     Writer
}
```

الإعلانات الثلاثة أعلاه لهم نفس التأثير، علمًا أن الترتيب غير مهم إنما المهم هو مجموعة الطرق.

تمرين 7.4: تقوم الدالة strings.NewReader باسترجاع قيمة تلبي واجهة io.Reader وغيرها من الواجهات من خلال القراءة من معاملاتها (سلسلة)، قم بتطبيق نسخة بسيطة من NewReader بنفسك واستخدمها لجعل محلل لغة توصيف النص التشعبي HTML (راجع القسم 5.2) يأخذ مدخله من سلسلة نصية.

تمرين 7.5: الدالة LimitReader من حزمة io تقبل io.Reader r وعدد من البايت n وترجع Reader آخر ليقرأ من r ويبلغ عن شرط نهاية الملف بعد n بايت، قم بتطبيق البرنامج.

7.3 إرضاء الواجهات Interface Satisfaction

يرضي نوع ما واجهة عندما يملك جميع الطرق التي تطلبها الواجهة، على سبيل المثال يرضي النوع os.File* الواجهات io.Reader و Writer و Closer و ReadWriter، ويلبي النوع bytes.Buffer* الواجهات Reader و Writer و ReadWriter ولكنه لا يرضي الواجهة Closer لأنه لا يملك طريقة Close. وللاختصار غالبًا يقول مبرمجو جو إن النوع المحدد عبارة عن نوع واجهة معينة، والمقصود بذلك أنه يرضي الواجهة، على سبيل المثال عندما يقول مبرمجو جو: bytes.Buffer* هو عبارة عن io.Writer، فهم يقصدون أن النوع bytes.Buffer* يرضي الواجهة io.Writer، وعندما يقولوا: os.File* عبارة عن io.ReadWriter، فيقصدون أن النوع os.File* يلبي الواجهة io.ReadWriter.

قاعدة التعيين (انظر 2.4.2) للواجهات بسيطة جدًا وهي أنه يمكن تعيين تعبير للواجهة فقط في حال كان نوع التعبير يرضي الواجهة، وبالتالي:

```
var w io.Writer
w = os.Stdout            // OK: *os.File has Write method
w = new(bytes.Buffer)         // OK: *bytes.Buffer has Write method
w = time.Second          // compile error: time.Duration lacks Write method

var rwc io.ReadWriteCloser
rwc = os.Stdout          // OK: *os.File has Read, Write, Close methods
rwc = new(bytes.Buffer)  // compile error: *bytes.Buffer lacks Close method
```

وتنطبق هذه القاعدة أيضًا عندما يكون الجانب الأيمن عبارة عن واجهة بنفسه:

```
w = rwc    // OK: io.ReadWriteCloser has Write method
rwc = w    // compile error: io.Writer lacks Close method
```

ولأن ReadWriter و ReadWriteCloser تشملان على جميع طرق Writer فأي نوع يرضي ReadWriter أو ReadWriteCloser يجب أن يرضي Writer.

قبل الخوض في التفاصيل سنشرح المقصود بحيازة النوع على طريقة، تعلمنا في القسم 6.2 أن كل نوع محدد مسمى T تمتلك بعض طرقه مستقبل من النوع T نفسه بينما آخرون يتطلبون مؤشر T*، وتذكر أنه يمكنك استدعاء طريقة T* لمعطيات النوع T طالما المعطى عبارة عن متغير، وبالتالي يأخذ المترجم عناوينه ضمنيًا ولكن هذا مجرد تجميل نحوي فقيمة النوع T لا تملك جميع الطرق التي يمتلكها المؤشر T* وبالتالي قد يرضي واجهات أقل.

دعنا نأخذ مثالًا للتوضيح، طريقة String للنوع IntSet (كما هو موجود في القسم 6.5) تتطلب مستقبل مؤشر وبالتالي لا يمكننا استدعاء الطريقة بقيمة IntSet غير موجهة:

```
type IntSet struct { /* ... */ }
func (*IntSet) String() string
var _ = IntSet{}.String() // compile error: String requires *IntSet receiver
```

ولكن يمكننا استدعائها على متغير IntSet:

```
var s IntSet
var _ = s.String() // OK: s is a variable and &s has a String method
```

وبما أن IntSet* الوحيدة التي تملك طريقة String فإن IntSet* فقط ترضي واجهة fmt.Stringer :

```
var _ fmt.Stringer = &s // OK
var _ fmt.Stringer = s // compile error: IntSet lacks String method
```

يوجد في القسم 12.8 برنامج يطبع طرق لقيمة كيفية (عشوائية) وتقوم الأداة godoc -analysis=type (راجع القسم 10.7.4) بعرض الطرق لكل نوع والعلاقة بين الواجهات والأنواع المحددة.

يمكن تشبيه ذلك بالمغلف الذي يحيط بالرسالة ويخفيها بداخله، فالواجهات تحيط النوع المحدد والقيمة وتخفيهما بداخلها. يمكن استدعاء فقط الطرق التي يكشفها نوع الواجهة حتى لو احتوى النوع المحدد على طرق أخرى:

```
os.Stdout.Write([]byte("hello")) // OK: *os.File has Write method
os.Stdout.Close()                // OK: *os.File has Close method
var w io.Writer
w = os.Stdout
w.Write([]byte("hello")) // OK: io.Writer has Write method
w.Close()                // compile error: io.Writer lacks Close method
```

تخبرنا الواجهات التي فيها طرق أكثر مثل io.ReadWriter عن المزيد من القيم التي تحتويها وعن الأماكن الأكثر طلبًا للأنواع التي تطبقها أكثر مما تخبرنا عنه الواجهات التي فيها طرق أقل مثل io.Reader، وبالتالي ماذا يخبرنا النوع interface{} عن الأنواع المحددة التي ترضيه علما أن النوع interface{} لا يحتوي على طرق على الإطلاق؟

أجل هذا صحيح: لا شيء، فالنوع interface{} لا يحتوي على أي طريقة، وبالتالي يبدو أنه عديم الفائدة! ولكن في الواقع النوع interface{} ويسمى أحيانًا بنوع الواجهة الفارغة ضروري ولا غنى عنه لأن نوع الواجهة الفارغة لا يتطلب أنواع ترضيه وبالتالي يمكننا تعيين أي قيمة للواجهة الفارغة.

```
var any interface{}
any = true
any = 12.34
any = "hello"
any = map[string]int{"one": 1}
any = new(bytes.Buffer)
```

قد تلاحظ أننا كنا نستخدم نوع الواجهة الفارغة منذ أول مثال في الكتاب وذلك لأنه يتيح للوظائف مثل fmt.Println أو errorf كما هو موضح في القسم 5.7 قبول معطيات من أي نوع.

وبالطبع عند إنشاء قيمة interface{} تحتوي على نوع boolean أو float أو string أو map أو pointer أو غيرها من الأنواع فإننا لا نستطيع فعل أي شيء مباشر بالقيمة التي يحملها لأن الواجهة لا تحتوي على طرق، وبالتالي نحتاج إلى طريقة لاسترجاع القيمة مرة أخرى، سنتعلم كيفية فعل ذلك باستخدام نوع التوكيد في القسم 7.10.

وبما أن إرضاء الواجهة يعتمد فقط على طرق النوعين المعنيين فلا يوجد هناك الحاجة لإعلان العلاقة بين النوع المحدد والواجهات التي يرضيها، فمن المفيد توثيق وتأكيد العلاقة عند استهدافها وليس إجبارها عبر البرنامج. الإعلان أدناه يؤكد في وقت الترجمة أن قيمة النوع *bytes.Buffer يرضي io.Writer:

```
// *bytes.Buffer must satisfy io.Writer
var w io.Writer = new(bytes.Buffer)
```

لم نحتاج لتخصيص أي متغير جديد كون أي قيمة من النوع bytes.Buffer* يتم تخصيصها حتى الصفر nil، علما أن nil تكتب هكذا (bytes.Buffer)(nil*) من خلال تحويل مباشر، وبما أننا لا ننوي أبدًا الإشارة إلى w فإننا يمكننا استبدالها بمعرف فارغ، وبالتالي كلا التغييرين يشكلان هذا البديل المبسط:

```
// *bytes.Buffer must satisfy io.Writer
var _ io.Writer = (*bytes.Buffer)(nil)
```

غالبًا ما ترضى أنواع الواجهات غير الفارغة مثل io.Writer بواسطة نوع مؤشر وخصوصًا عندما تتضمن طريقة أو أكثر من طرق الواجهات نوعا من التغيير للمستقبل كما تفعل طريقة Write، ومن أكثر أنواع محملات الطرق شيوعًا هو المؤشر إلى البنية.

ولكن أنواع المؤشرات ليس هي الأنواع الوحيدة التي تُرضي الواجهات، حتى مع الواجهات ذات طرق مُغيِّرة يمكن إرضاءها بواسطة أحد أنواع المرجعية الأخرى في لغة جو. لقد عرضنا عدة أمثلة عن أنواع الشرائح وطرقها (geometry.Path في القسم 6.1) وعرضنا أمثلة عن أنواع الخرائط وطرقها (url.Values في القسم 6.2.1)، ولاحقًا سنرى أنواع الوظائف وطرقها(http.HandlerFunc في القسم 7.7)، وكما سنرى في القسم 7.4 فإن العديد من الأنواع الأساسية قد ترضي الواجهات فإن time.Duration ترضي fmt.Stringer.

قد يرضي نوع محدد العديد من الواجهات ليست ذات الصلة، فكر في برنامج ينظم أو يبيع فنون ثقافية رقمية مثل الموسيقى والأفلام والكتب، فقد يعرف مجموعة الأنواع المحددة التالية:

ألبوم Album

كتاب Book

فلم Movie

مجلة Magazine

نشرة (صوتية)Podcast

حلقات مسلسلات TVEpisode

تسجيل صوتي (أغنية)Track

يمكننا التعبير عن كل تجريد مهم بواجهة، وبعض الخصائص مشتركة بين جميع الفنون مثل العنوان وتاريخ الإنشاء وقائمة المبتكرين (المؤلف أو الفنان).

```
type Artifact interface {
	Title() string
	Creators() []string
	Created() time.Time
}
```

وهناك خصائص أخرى محددة لأنواع محددة من الفنون، فخاصية الكلمات المطبوعة مرتبطة فقط في الكتب والمجلات بينما دقة الشاشة مرتبطة بالأفلام والمسلسلات.

```
type Text interface {
	Pages() int
	Words() int
	PageSize() int
}
type Audio interface {
	Stream() (io.ReadCloser, error)
	RunningTime() time.Duration
	Format() string // e.g., "MP3", "WAV"
}
type Video interface {
	Stream() (io.ReadCloser, error)
	RunningTime() time.Duration
	Format() string // e.g., "MP4", "WMV"
	Resolution() (x, y int)
}
```

فمن الجيد استخدام تلك الواجهات لتجميع الأنواع المحددة المترابطة معًا والتعبير عن الجوانب التي يشتركون فيها معًا، قد نكتشف عن تجمعات أخرى لاحقًا، على سبيل المثال إذا احتجنا للتعامل مع الصوت والفيديو بنفس الطريقة فسنقوم بتعريف واجهة Streamer لتمثيل جوانبهم المشتركة دون التغيير في إعلان أي نوع موجود.

```
type Streamer interface {
	Stream() (io.ReadCloser, error)
	RunningTime() time.Duration
	Format() string
}
```

يمكن التعبير عن كل مجموعة من الأنواع المحددة المشتركة في السلوكيات بنوع واجهة، وعلى النقيض من لغات البرمجة كائنية التوجه التي ترضى الواجهات فيها من خلال الصف مباشرة ولكن في لغة البرمجة جو يمكننا تعريف تجريدات أو مجموعات جديدة دون التعديل على إعلان الأنواع المحددة، وهذا مفيد عندما نأخذ النوع المحدد من حزمة مكتوبة بواسطة مؤلفين مختلفين، وبالطبع يجب على المؤلفين تحديد الأساسيات المشتركة بين الأنواع المحددة.

7.4 تحليل الأعلام باستخدام الواجهة (flag.Value)

(توضيح: تستخدم الجيوش الأعلام لإرسال رسالة معينة طبقًا لحركات معينة في العلم، وكذلك في لغة البرمجة تستخدم الأعلام لتوصيل رسالة أو معلومة معينة لذلك سميت بالأعلام).

سنتعلم في هذا الفصل عن واجهة قياسية جديدة وهي flag.Value، حيث تساعدنا هذه الواجهة على تعريف رموز جديدة لأعلام سطر الأوامر. انظر للبرنامج أدناه حيث يتوقف (ينام) لفترة زمنية معينة:

```
gopl.io/ch7/sleep
var period = flag.Duration("period", 1*time.Second, "sleep period")
func main() {
    flag.Parse()
    fmt.Printf("Sleeping for %v...", *period)
    time.Sleep(*period)
    fmt.Println()
}
```

يقوم البرنامج بطباعة المدة الزمنية التي سينام فيها قبل الذهاب للنوم، فتقوم الحزمة fmt باستدعاء طريقة String التابعة لـ time.Duration لطباعة الفترة ليس بوحدة (nanoseconds) (واحد من المليار من الثانية) ولكن برموز مفهومة للمستخدم:

```
$ go build gopl.io/ch7/sleep
$ ./sleep
Sleeping for 1s...
```

في الوضع الافتراضي تكون مدة النوم ثانية واحدة فقط، ولكن يمكن التحكم بها من خلال علم سطر الأوامر period- ، تقوم الدالة flag.Duration بإنشاء متغير علم من النوع time.Duration مما يتيح للمستخدم إمكانية تحديد المدة الزمنية بصيغ مفهومة متعددة بما فيها الرموز المطبوعة بواسطة الطريقة String، عند جعل التصميم متناسق كالموجود أدناه فإنه يجعل واجهة المستخدم جميلة.

```
$ ./sleep -period 50ms
Sleeping for 50ms...
$ ./sleep -period 2m30s
Sleeping for 2m30s...
$ ./sleep -period 1.5h
Sleeping for 1h30m0s...
$ ./sleep -period "1 day"
invalid value "1 day" for flag -period: time: invalid duration 1 day
```

ولأن الأعلام التي لديها قيمة للمدة الزمنية مفيدة للمستخدم، فقد وضعت في حزمة الأعلام flag ولكن من السهل تعريف رموز جديدة للعلم للأنواع الخاصة بنا، نحتاج فقط لتعريف النوع الذي يرضي الواجهة flag.Value والذي إعلانه موضح أدناه:

```
package flag
// Value is the interface to the value stored in a flag.
type Value interface {
     String() string
     Set(string) error
}
```

تقوم طريقة String بصياغة قيمة العلم لاستخدامها في رسالة المساعدة لسطر الأوامر، وبالتالي يعتبر كل flag.Value مثل fmt.Stringer، وتقوم الطريقة Set بتحليل معطيات سلسلتها وتحديث قيمة العلم. في الواقع تعتبر طريقة Set معكوس طريقة String، في حال رغبت بالتمرن على كلا الطريقتين استخدم نفس الترميز.

دعنا نعرف نوع علم celsiusFlag الذي بدوره يتيح تحديد الحرارة بالدرجة المئوية Celsius، أو يمكن تمثيله بدرجة فهرنهايت Fahrenheit من خلال إجراء التحويل اللازم، لاحظ أن celsiusFlag يضمن Celsius (راجع القسم 2.5) وبالتالي طريقة String جاهزة تلقائيًا، ومن أجل إرضاء flag.Value يجب علينا إعلان طريقة Set:

```
gopl.io/ch7/tempconv
// *celsiusFlag satisfies the flag.Value interface.
type celsiusFlag struct{ Celsius }
func (f *celsiusFlag) Set(s string) error {
   var unit string
   var value float64
   fmt.Sscanf(s, "%f%s", &value, &unit) // no error check needed
   switch unit {
   case "C", "°C":
      f.Celsius = Celsius(value)
      return nil
   case "F", "°F":
      f.Celsius = FToC(Fahrenheit(value))
      return nil
   }
   return fmt.Errorf("invalid temperature %q", s)
}
```

استدعاء fmt.Sscanf يقوم بتحليل رقم النقطة العائمة floating point (قيمة value) وسلسلة (unit) من المدخل s، ويفضل غالبًا تفقد أخطاء نتائج Sscanf ولكن في هذه الحالة لا يوجد الحاجة لذلك لأنه إذا وجد مشكلة ما لن تتطابق المبدلات.

تحتوي وظيفة CelsiusFlag على كل شيء كما هو موضح أدناه، وبالنسبة للمستدعي يقوم بإرجاع المؤشر إلى حقل Celsius المضمن داخل المتغير f الخاص بـ celsiusFlag، حقل Celsius عبارة عن متغير يحدث بواسطة طريقة Set خلال

معالجة الأعلام، وعند استدعاء Var يضاف العلم إلى مجموعة تطبيقات أعلام سطر الأوامر، ألا وهو المتغير العام flag.CommandLine. قد تحتوي البرامج ذات الواجهات المعقدة على عدة متغيرات لهذا النوع، فاستدعاء Var يعين معطيات‪*celsiusFlag‬ لمعاملات flag.Value مما يجعل المترجم يتفقد أن ‪*celsiusFlag‬ لديه الطرق اللازمة.

```
// CelsiusFlag defines a Celsius flag with the specified name,
// default value, and usage, and returns the address of the flag variable.
// The flag argument must have a quantity and a unit, e.g., "100C".
func CelsiusFlag(name string, value Celsius, usage string) *Celsius {
    f := celsiusFlag{value}
    flag.CommandLine.Var(&f, name, usage)
    return &f.Celsius
}
```

يمكننا الآن استخدام العلم الجديد في برنامجنا:

```
gopl.io/ch7/tempflag
var temp = tempconv.CelsiusFlag("temp", 20.0, "the temperature")
func main() {
    flag.Parse()
    fmt.Println(*temp)
}
```

وها هنا الجلسة النموذجية:

```
$ go build gopl.io/ch7/tempflag
$ ./tempflag
20°C
$ ./tempflag -temp -18C
-18°C
$ ./tempflag -temp 212°F
100°C
$ ./tempflag -temp 273.15K
invalid value "273.15K" for flag -temp: invalid temperature "273.15K"
Usage of ./tempflag:
    -temp value
          the temperature (default 20°C)
$ ./tempflag -help
Usage of ./tempflag:
    -temp value
          the temperature (default 20°C)
```

تمرين 7.6: قم بإضافة حرارة Kelvin إلى tempflag.

تمرين 7.7: فسر لماذا تحتوي رسالة المساعدة على C° في حين أن القيمة الافتراضية 20.0 لا تحتوي على C°.

7.5 قيم الواجهات Interface Values

تتكون قيمة نوع الواجهة من الناحية النظرية من عنصرين وهما النوع المحدد وقيمة ذلك النوع، ويطلق عليهما اسم النوع الديناميكي (dynamic type) والقيمة الديناميكية (dynamic value) للواجهة.

بالنسبة للغات التي تستخدم أنواع ثابتة (مثل لغة جو)، المقصود بالنوع هو مفهوم وقت التحويل البرمجي أو وقت المترجم، وبالتالي لا يعتبر النوع كقيمة. مجموعة القيم تسمى واصفات النوع كونها تعطي معلومات عن كل نوع مثل اسمه وطرقه، بالنسبة لقيمة الواجهة فإن عنصر النوع يتمثل في وصف لائق للنوع.

في الجمل البرمجية الأربعة أدناه يأخذ المتغير w ثلاثة قيم مختلفة. (القيمة البدائية والقيمة النهائية متشابهتان).

```
var w io.Writer
w = os.Stdout
w = new(bytes.Buffer)
w = nil
```

لنركز قليلًا على قيمة w وسلوكها الديناميكي لكل جملة، الجملة الأولى تعلن w:

```
var w io.Writer
```

دائمًا تتم تهيئة المتغيرات في لغة جو بقيمة محددة جيدًا، وكذلك الحال مع الواجهات، فالواجهات تحتوي على قيمة صفر nil ليمثل كلا عنصري النوع والقيمة (انظر إلى الصورة 7.1).

w	
type	nil
value	nil

صورة 7.1: واجهة قيمتها صفر

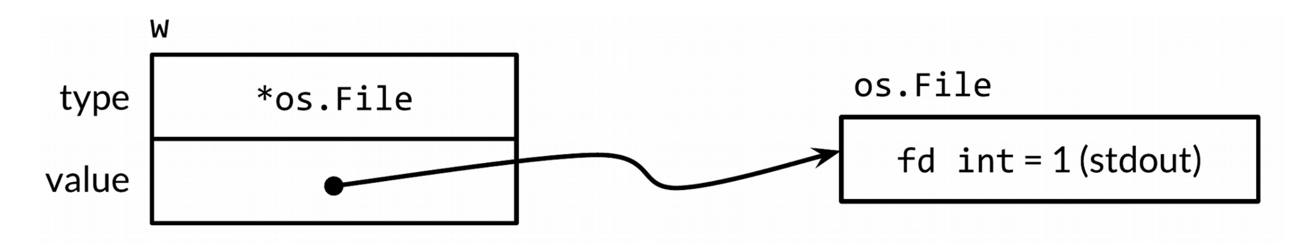
توصف قيمة الواجهة بصفر nil أو غير الصفر non-nil طبقًا لنوعها الديناميكي، وبالتالي هذه واجهة ذات قيمة صفر، بإمكانك فحص قيمة الواجهة إذا كانت صفر من خلال استخدام w == nil أو w != nil، وفي حال استدعيت أي طريقة من واجهة قيمتها صفر سيتسبب في هلع:

```
w.Write([]byte("hello")) // panic: nil pointer dereference
```

تحدد الجملة الثانية قيمة في w من النوعos.File* :

```
w = os.Stdout
```

هذا التعيين يشمل على تحويل ضمني من نوع محدد إلى نوع واجهة وهو مشابه للتحويل المباشر io.Writer(os.Stdout)، سواء كان التحويل ضمني أو مباشر فإنه يحتجز نوع وقيمة معامله، النوع الديناميكي لقيمة الواجهة يعين لواصف النوع لمؤشر من النوع os.File* وتحمل قيمته الديناميكية نسخة من os.Stdout والتي هي عبارة عن مؤشر للمتغير os.File ليمثل المخرج القياسي للعملية (انظر إلى الصورة 7.2).



صورة 7.2: قيمة الواجهة تحتوي على مؤشر os.File*

عند استدعاء طريقة Write في قيمة واجهة تحتوي على مؤشر os.File* تتسبب في استدعاء طريقة (os.File*).Write، ليطبع الاستدعاء "hello".

```
w.Write([]byte("hello")) // "hello"
```

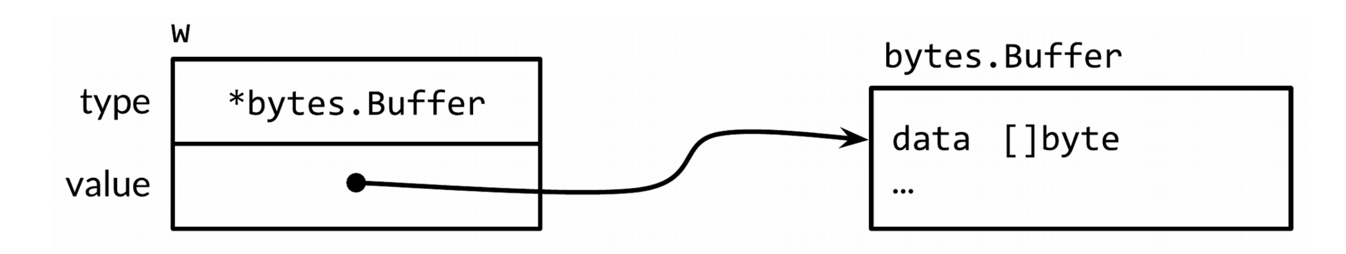
بشكل عام لا يمكننا معرفة النوع الديناميكي لقيمة الواجهة حسب وقت المترجم، وبالتالي يجب الاستدعاء عبر الواجهة أن يستخدم إيفاد ديناميكي (dynamic dispatch)، وبالتالي بدلًا من الاستدعاء المباشر يقوم المترجم بإنشاء كود لإيجاد عنوان الطريقة المسمية بـ Write من واصف النوع وثم يستدعي ذلك العنوان بشكل غير مباشر. ومعطيات المستقبل عبارة عن نسخة من القيمة الديناميكية للواجهة os.Stdout، والتأثير مشابه لحالة الاستدعاء المباشر:

```
os.Stdout.Write([]byte("hello")) // "hello"
```

تعين الجملة الثالثة قيمة للنوع bytes.Buffer* لقيمة الواجهة:

```
w = new(bytes.Buffer)
```

وبالتالي النوع الديناميكي هو bytes.Buffer* والقيمة الديناميكية هي مؤشر إلى المخزن المؤقت المحدد حديثًا (أنظر إلى الصورة 7.3).



صورة 7.3: قيمة الواجهة تحتوي على مؤشر *bytes.Buffer

استدعاء طريقة Write يستخدم نفس الآليات السابقة:

```
w.Write([]byte("hello")) // writes "hello" to the bytes.Buffer
```

في هذه المرة واصف النوع هو *bytes.Buffer وبالتالي تم استدعاء الطريقة (*bytes.Buffer).Write وعنوان المخزن المؤقت هو نفس قيمة معامل المستقبل، فالاستدعاء يلحق "hello" للمخزن المؤقت.

أخيرا تقوم الجملة الرابعة بتعيين nil لقيمة الواجهة:

```
w = nil
```

هذا يسبب في إعادة تعيين كل من عناصره إلى nil وإعادة w إلى وضعها السابق عندما أعلنت كما وُضِّحَ في الصورة 7.1.

قد تحمل قيمة الواجهة قيم ديناميكية كبيرة عشوائية، على سبيل المثال النوع time.Time يمثل لحظة من الزمن وهو عبارة عن نوع بنية بحقول غير مُصَدّرَة، فإذا أنشأنا قيمة واجهة منه:

```
var x interface{} = time.Now()
```

قد تظهر النتائج كما هو موضح في الصورة 7.4، غالبًا تتناسب القيمة الديناميكية داخل قيمة الواجهة بغض النظر عن كبر حجم نوعه، (هذا مجرد نموذج نظري تصوري، ولكن التطبيق الفعلي يختلف قليلًا)

x
type
value
time.Time
sec: 63567389742
nsec: 689632918
loc: “UTC”

صورة 7.4: قيمة الواجهة تحمل البنية time.Time

يمكن مقارنة قيم الواجهة بواسطة == و !=، تتساوى قيمتي الواجهة في حال كانتا كلتاهما nil أو نوعيهما الديناميكيين متماثلان وقيمتهما الديناميكيتين متساويتين طبقًا للسلوك العام لـ == لذلك النوع. وبما أن قيم الواجهة قابلة للمقارنة فيمكن استخدامهم كمفاتيح لخريطة أو كمعامل لجملة switch.

ولكن في حالة مقارنة قيمتين للواجهة وكان لهما نفس النوع الديناميكي ولكن ذلك النوع غير قابل للمقارنة (مثل الشريحة slice) فإن عملية المقارنة ستفشل مع هلع:

```
var x interface{} = []int{1, 2, 3}
fmt.Println(x == x) // panic: comparing uncomparable type []int
```

في هذه الحالة تكون أنواع الواجهات استثنائية، فالأنواع عادة تكون إما قابلة للمقارنة بشكل آمن (مثل الأنواع والمؤشرات الأساسية) أو غير قابلة إطلاقا للمقارنة (مثل الشرائح والخرائط والوظائف) ولكن عند مقارنة قيم واجهة أو أنواع مجمعة تحتوي على قيم واجهة فيجب علينا الانتباه لوجود الأخطاء، وقد تواجه مثل هذه الخطورة عند استخدام الواجهات كمفتاح خارطة أو معامل إبدال. قارن فقط قيم الواجهات التي تحتوي على أنواع ديناميكية لأنواع قابلة للمقارنة.

عند التعامل مع الأخطاء وتصحيحها قم بالتبليغ عن النوع الديناميكي لقيمة الواجهة، ولهذا الغرض نستخدم %T الخاصة بحزمة fmt :

```
var w io.Writer
fmt.Printf("%T\n", w) // "<nil>"

w = os.Stdout
fmt.Printf("%T\n", w) // "*os.File"
w = new(bytes.Buffer)
fmt.Printf("%T\n", w) // "*bytes.Buffer"
```

تقوم fmt باستخدام الانعكاس (reflection) داخليًا لإيجاد النوع الديناميكي للنوع، سنشرح الانعكاس في الفصل الثاني عشر.

7.5.1 تنبيه: الواجهة التي تحتوي على مؤشر Nil تعتبر واجهة ذات قيمة غير صفرية Non-Nil

الواجهة التي تكون قيمتها صفر (أو لا شيء) تختلف عن الواجهة التي تحتوي قيمتها على مؤشر صفر (أو لا شيء)، هذا التمييز الدقيق يخلق فخا يقع فيه معظم مبرمجي جو.

انظر إلى البرنامج أدناه، عندما يكون التصحيح true تقوم الدالة main بجمع مخارج الدالة f في bytes.Buffer:

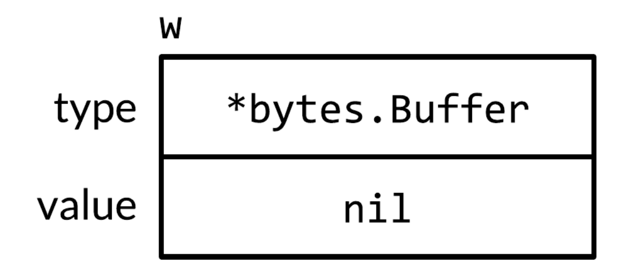
```
const debug = true

func main() {
    var buf *bytes.Buffer
    if debug {
            buf = new(bytes.Buffer) // enable collection of output
    }
f(buf) // NOTE: subtly incorrect!
if debug {
            // ...use buf...
    }
}
// If out is non-nil, output will be written to it.
func f(out io.Writer) {
    // ...do something...
    if out != nil {
            out.Write([]byte("done!\n"))
    }
}
```

وقد نلاحظ أن تغير التصحيح إلى false سيتسبب في تعطيل تجميع المخرج ولكن في الواقع يسبب في فشل البرنامج خلال استدعاء out.Write:

```
if out != nil {
    out.Write([]byte("done!\n")) // panic: nil pointer dereference
}
```

عندما تقوم main باستدعاء f فإنها تعين مؤشر nil من النوع bytes.Buffer* لمعاملات المخرج وبالتالي تكون القيمة الديناميكية للمخرج nil، ونوعه الديناميكي هو bytes.Buffer* مما يعني أن المخرج عبارة عن واجهة non-nil تحتوي على قيمة مؤشر nil (انظر إلى الصورة 7.5) وبالتالي ما زال التحقق nil =! هو true.



صورة 7.5: واجهة non-nil تحتوي على مؤشر nil

كما في السابق تحدد آلية الإيفاد الديناميكي أنه يجب استدعاء bytes.Buffer).Write*) ولكن في هذه المرة ستكون قيمة المستقبل nil، في بعض الأنواع مثل os.File* تكون nil مستقبلًا مقبولًا (راجع القسم 6.2.1) ولكن لا يعتبر bytes.Buffer* من ضمنهم. يتم استدعاء الطريقة ولكن يحدث خطأ عندما تحاول الوصول إلى المخزن المؤقت.

المشكلة هي أنه بالرغم من حصول المؤشر bytes.Buffer* ذي القيمة nil على جميع الطرق اللازمة لإرضاء الواجهة إلا أنه لا يرضي المتطلبات السلوكية للواجهة. وبالأخص عملية الاستدعاء تخالف الشرط الضمني لـ (bytes.Buffer*). (Write وهو ألا يكون المستقبل nil، وبالتالي من الخطأ تعيين مؤشر nil للواجهة. الحل هو تغيير النوع لـ buf في main إلى io.Writer وبالتالي تجنب تعيين القيمة المختلة وظيفيًا في الواجهة:

```
var buf io.Writer
if debug {
    buf = new(bytes.Buffer) // enable collection of output
}
f(buf) // OK
```

ولآن بما أننا شرحنا مبدأ قيم الواجهات دعونا نتطرق إلى بعض الواجهات المهمة في مكتبة جو القياسية، في الأقسام الثلاثة التالية سنتعلم عن كيفية استخدام الواجهات في الفرز وخدمة الإنترنت والتعامل مع الأخطاء.

7.6 الفرز باستخدام sort.Interface

الفرز أو الترتيب (sorting) هو عملية شائعة الاستخدام في مختلف البرامج، ولكن يحتاج لكتابة أسطر كثيرة فالترتيب السريع يحتاج لكتابة 15 سطر تقريبًا والتطبيقات المعقدة والمتينة تحتاج لأسطر أكثر، فمن الممل كتابته كل مرة نحتاج إلى الفرز أو الترتيب.

لحسن الحظ، تقدم حزمة الفرز sort الترتيب الموضعي بأي تسلسل نرغب به، فتصميمها استثنائي. في العديد من لغات البرمجة تكون خوارزمية الترتيب مرتبطة بنوع بيانات المتسلسلة، في حين أن دالة الترتيب مرتبطة بنوع العناصر، وعلى النقيض من ذلك في لغة جو لا تفترض الدالة sort.Sort أي شيء بشأن تمثيل سواء كان التسلسل أو العناصر، عوضا عن ذلك، تستخدم الواجهة sort.Interface لتحديد العقد بين خوارزمية الترتيب العامة وكل نوع تسلسل ترغب بترتيبه، تطبيق هذه الواجهة يحدد كلا التمثيل المحدد للتسلسل (والذي غالبًا يكون شريحة) والترتيب المطلوب لعناصرها.

تحتاج خوارزمية الترتيب الموضعي إلى ثلاثة أشياء: طول التسلسل والطرق المستخدمة للمقارنة بين عنصرين وطريقة استبدال العنصرين ببعضهما، وبالتالي هم الطرق الثلاثة لـ sort.Interface:

```
package sort
type Interface interface {
    Len() int
    Less(i, j int) bool // i, j are indices of sequence elements
    Swap(i, j int)
}
```

ولترتيب أي تسلسل نحتاج إلى تحديد نوع يحقق الطرق الثلاثة ثم تطبيق sort.Sort على واجهة من ذلك النوع، على سبيل المثال نريد ترتيب شريحة من السلاسل النصية، سيكون النوع الجديد هو StringSlice والطرق الثلاثة هي Len و Less و Swap كما هي موضحة أدناه.

```
type StringSlice []string

func (p StringSlice) Len() int           { return len(p) }
func (p StringSlice) Less(i, j int) bool  { return p[i] < p[j] }
func (p StringSlice) Swap(i, j int)       { p[i], p[j] = p[j], p[i] }
```

الآن يمكننا ترتيب شريحة من السلاسل النصية، names ، من خلال تحويل الشريحة إلى StringSlice هكذا:

```
sort.Sort(StringSlice(names))
```

نتيجة التحويل هي الحصول على قيمة شريحة لها مصفوفة بنفس طول وسعة والتمثيل الداخلي لـ names ولكن بنوع يمتلك ثلاثة طرق للترتيب.

ترتيب شريحة من السلاسل أمرٌ شائع وبالتالي يتوفر النوع StringSlice في حزمة sort ، وكذلك تتوفر الدالة Strings وبالتالي يمكن تبسيط الاستدعاء أعلاه إلى sort.Strings(names).

ويمكن ملاءمة الطريقة نفسها لترتيبات أخرى، على سبيل المثال تجاهل الأحرف الكبيرة أو الأحرف الخاصة (مثل برنامج جو الذي يرتب فهرس المصطلحات وترقيم الصفحات لهذا الكتاب مع إضافة منطقية للأرقام الرومانية)، نستخدم نفس الفكرة للترتيبات المعقدة ولكن في المقابل سيزداد تعقيد بنية البيانات والتطبيقات لطرق sort.Interface .

سيكون المثال التالي عن الترتيب هو قائمة تشغيل أغاني playlist معروضة في جدول، كل أغنية track موجودة في سطر وفي كل عمود يوجد صفات تلك الأغنية مثل اسم الفنان والعنوان ومدة الأغنية، تخيل وجود واجهة مستخدم بيانية تمثل الجدول، وعند النقر على راس العمود يقوم بترتيب القائمة حسب صفات الأغنية، وعند النقر مرة أخرى على راس العمود يقوم بعكس الترتيب، دعنا نرى ماذا سيحصل عند كل نقرة.

يحتوي المتغير tracks أدناه على قائمة التشغيل، وكل متغير غير مباشر أي مؤشر إلى أغنية، ولكن الكود أدناه يعمل في حال قمنا بترتيب الأغاني مباشرة، أي ستقوم الدالة sort باستبدال العديد من أزواج العناصر وبالتالي سيعمل بشكل أسرع في حال كان كل عنصر كمؤشر وهو عبارة عن كلمة آلية واحدة بدلًا من استخدام اسم الأغنية كاملًا كونه قد تصل اسم الأغنية إلى 8 كلمات أو أكثر.

```
type Track struct {
   Title  string
   Artist string
   Album  string
   Year   int
```

```
    Length time.Duration
}
var tracks = []*Track{
    {"Go", "Delilah", "From the Roots Up", 2012, length("3m38s")},
    {"Go", "Moby", "Moby", 1992, length("3m37s")},
    {"Go Ahead", "Alicia Keys", "As I Am", 2007, length("4m36s")},
    {"Ready 2 Go", "Martin Solveig", "Smash", 2011, length("4m24s")},
}
func length(s string) time.Duration {
    d, err := time.ParseDuration(s)
    if err != nil {
        panic(s)
    }
    return d
}
```

تقوم الدالة printTracks بطباعة قائمة التشغيل على شكل جدول، سيكون جميلًا لو رسمت خطوط الجدول بشكل واضح ولكن للأسف هذا النمط يستخدم حزمة text/tabwriter ليعرض جدولًا تكون أعمدته مرتبة بشكل أنيق كما هو ظاهر أدناه، لاحظ أن tabwriter.Writer* ترضي io.Writer، إنها تجمع كل قطعة من البيانات مكتوبة إليها، تقوم طريقتها Flush بتنسيق كامل الجدول وكتابته إلى os.Stdout.

```
func printTracks(tracks []*Track) {
    const format = "%v\t%v\t%v\t%v\t%v\t\n"
    tw := new(tabwriter.Writer).Init(os.Stdout, 0, 8, 2, ' ', 0)
    fmt.Fprintf(tw, format, "Title", "Artist", "Album", "Year", "Length")
    fmt.Fprintf(tw, format, "-----", "------", "-----", "----", "------")
    for _, t := range tracks {
        fmt.Fprintf(tw, format, t.Title, t.Artist, t.Album, t.Year, t.Length)
    }
    tw.Flush() // calculate column widths and print table
}
```

لترتيب القائمة حسب اسم الفنان سنقوم بتعريف نوع شريحة جديد بالطرق Len و Less و Swap المناسبة ليماثل ما فعلناه مع StringSlice.

```
type byArtist []*Track
func (x byArtist) Len() int           { return len(x) }
func (x byArtist) Less(i, j int) bool { return x[i].Artist < x[j].Artist }
func (x byArtist) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
```

لاستدعاء نمط الترتيب العام سنقوم أولًا بتحويل الأغاني إلى نوع جديد وهو byArtist مما يعرف الترتيب:

```
sort.Sort(byArtist(tracks))
```

وبعد ترتيب الشريحة حسب اسم الفنان، سيكون مخرج printTracks كالتالي:

```
Title       Artist          Album               Year  Length
-----       ------          -----               ----  ------
```

```
Go Ahead    Alicia Keys      As I Am             2007  4m36s
Go          Delilah          From the Roots Up   2012  3m38s
Ready 2 Go  Martin Solveig   Smash               2011  4m24s
Go          Moby             Moby                1992  3m37s
```

في حال أراد المستخدم طلب ترتيب القائمة حسب اسم الفنان مرة أخرى فإن ترتيب الأغاني سينعكس، ولن نحتاج لتعريف نمط جديد معكوس byReverseArtist بطريقة Less معكوسة، وعلى كل حال، فإن حزمة الترتيب تحتوي على دالة العكس Reverse التي تقوم بعكس أي ترتيب.

```
sort.Sort(sort.Reverse(byArtist(tracks)))
```

بعد عكس ترتيب الشريحة حسب اسم الفنان ستكون نتيجة printTracks كالتالي:

```
Title       Artist           Album               Year  Length
-----       ------           -----               ----  ------
Go          Moby             Moby                1992  3m37s
Ready 2 Go  Martin Solveig   Smash               2011  4m24s
Go          Delilah          From the Roots Up   2012  3m38s
Go Ahead    Alicia Keys      As I Am             2007  4m36s
```

يجب علينا التركيز أكثر في الدالة sort.Reverse كونها تستخدم التركيب composition (راجع القسم 6.3) فهي فكرة مهمة، تحدد حزمة الترتيب نوع غير مُصدّر reverse وهو بنية ضمن sort.Interface، تستدعي طريقة Less للعكس طريقة Less لقيمة sort.Interface المضمنة ولكن مع عكس المؤشرات مما يعكس ترتيب النتائج.

```
package sort
type reverse struct{ Interface } // that is, sort.Interface
func (r reverse) Less(i, j int) bool { return r.Interface.Less(j, i) }
func Reverse(data Interface) Interface { return reverse{data} }
```

طريقتان الأخريتان Len و Swap تُقدمان بواسطة قيمة sort.Interface الأصلية لأنها حقل مضمن، تسترجع الدالة المصدرة Reverse نسخة من النوع reverse الذي يحتوي على قيمة sort.Interface الأصلية.

للترتيب حسب عمود مختلف يجب علينا تعريف نوع جديد مثل byYear:

```
type byYear []*Track
func (x byYear) Len() int           { return len(x) }
func (x byYear) Less(i, j int) bool { return x[i].Year < x[j].Year }
func (x byYear) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
```

بعد ترتيب الأغاني حسب السنة باستخدام sort.Sort(byYear(tracks)) ستطبع الوظيفة printTracks الجدول مرتب حسب التسلسل الزمني:

```
Title       Artist          Album             Year  Length
-----       ------          -----             ----  ------
Go          Moby            Moby              1992  3m37s
Go Ahead    Alicia Keys     As I Am           2007  4m36s
Ready 2 Go  Martin Solveig  Smash             2011  4m24s
Go          Delilah         From the Roots Up 2012  3m38s
```

نقوم بإعلان تطبيق جديد لـ sort.Interface لكل نوع عنصر شريحة وكل دالة ترتيب نحتاجها، وكما تلاحظ أن الطريقتين Len و Swap لهما تعريفات متماثلة لجميع أنواع الشريحة، في المثال التالي سترى أن النوع المحدد customSort يجمع شريحة مع دالة مما يتيح لنا إمكانية تعريف ترتيب جديد من خلال كتابة دالة المقارنة فقط، علمًا أن الأنواع المحددة التي تطبق sort.Interface ليست دائمًا شرائح، فمثلًا customSort هو نوع بنية.

```
type customSort struct {
    t    []*Track
    less func(x, y *Track) bool
}
func (x customSort) Len() int           { return len(x.t) }
func (x customSort) Less(i, j int) bool { return x.less(x.t[i], x.t[j]) }
func (x customSort) Swap(i, j int)      { x.t[i], x.t[j] = x.t[j], x.t[i] }
```

دعنا نعرف دالة ترتيب متعددة المستويات بحيث يكون مفتاح الترتيب الرئيسي لها هو Title ومفتاح الترتيب الثانوي هو السنة Year ومفتاح الترتيب الثالث هو الطول length، هنا نستدعي Sort باستخدام دالة ترتيب مجهولة:

```
sort.Sort(customSort{tracks, func(x, y *Track) bool {
    if x.Title != y.Title {
        return x.Title < y.Title
    }
    if x.Year != y.Year {
        return x.Year < y.Year
    }
    if x.Length != y.Length {
        return x.Length < y.Length
    }
    return false
}})
```

وستكون النتائج كما هي ظاهرة أدناه، لاحظ وجود أغنيتين اسمهما Go وبالتالي لا يمكن ترتيبهما حسب الاسم ولكن تغير ترتيبهما طبقًا للسنة.

```
Title       Artist          Album             Year  Length
-----       ------          -----             ----  ------
Go          Moby            Moby              1992  3m37s
Go          Delilah         From the Roots Up 2012  3m38s
Go Ahead    Alicia Keys     As I Am           2007  4m36s
Ready 2 Go  Martin Solveig  Smash             2011  4m24s
```

إن ترتيب تسلسل الطول n يتطلب عمليات مقارنة O(n log n) ولاختبار ما إذا يتطلب التسلسل الذي قد تم ترتيبه على الأقل (n−1) عملية مقارنة، ولكن لا يجب علينا القيام بذلك يدويًا لأنه يوجد دالة IsSorted في حزمة sort تقوم بذلك، حيث تقوم بتجريد كل من دالة الترتيب والتسلسل باستخدام sort.Interface ولكن لا يستدعي طريقة Swap، الكود التالي يوضح الدالتين IntsAreSorted و Ints والنوع IntSlice:

```
values := []int{3, 1, 4, 1}
fmt.Println(sort.IntsAreSorted(values))    // "false"
sort.Ints(values)
fmt.Println(values)                        // "[1 1 3 4]"
fmt.Println(sort.IntsAreSorted(values))    // "true"
sort.Sort(sort.Reverse(sort.IntSlice(values)))
fmt.Println(values)                        // "[4 3 1 1]"
fmt.Println(sort.IntsAreSorted(values))    // "false"
```

ولتسهيل الأمور تزودنا حزمة الترتيب بإصدارات متنوعة من وظائفها وأنواعها المخصصة لـ []int و[]string و[]float64 باستخدام ترتيبهم الطبيعي، ولكن يجب عليك الاعتماد على نفسك في بعض الأنواع مثل[]int64 أو[]uint علمًا أن المسار قصيرًا.

تمرين 7.8: تعرض لك العديد من واجهات المستخدم الرسومية جدولًا مناسبًا للترتيب متعدد المستويات، ويكون مفتاح الترتيب الرئيسي هو آخر عمود نقر على رأسه، ومفتاح الترتيب الثانوي هو ثاني آخر عمود نقر على رأسه وهكذا، عرّف تطبيق لـ sort.Interface لاستخدامه في مثل هذه الجداول، وقارن ذلك النهج مع الترتيب المتكرر باستخدام sort.Stable.

تمرين 7.9: استخدم الحزمة html/template (راجع القسم 4.6) لاستبدال printTracks مع دالة تعرض الأغاني في جدول HTML، واستخدم الحل في التمرين السابق للترتيب بحيث كل نقرة على رأس العمود تجعل HTML يطلب ترتيب الجدول.

تمرين 7.10: يمكن الاستفادة من النوع sort.Interface في استخدامات أخرى، اكتب دالة منطقية

IsPalindrome(s sort.Interface) بحيث ترجع إذا ما كان التسلسل s متناظرًا، أي بمعنى آخر عند عكس التسلسل لن يغيره، افترض أن العناصر عند المؤشرين i و j متساويان في حال الشرط !s.Less(i, j) && !s.Less(j, i).

7.7 الواجهة http.Handler

أخذنا في الفصل الأول لمحة عن كيفية استخدام الحزمة net/http لتطبيقات عملاء الوب (راجع القسمين 1.5 والخادم (راجع 1.7)، وفي هذا القسم سنتعلم المزيد عن دوال API الخاصة بالخادم والذي أساسها حزمة http.Handler:

```go
net/http
package http
type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}
func ListenAndServe(address string, h Handler) error
```

تتطلب الدالة ListenAndServe عنوان خادم مثل "localhost:8000" وحالة واجهة Handler التي تحول إليها جميع الطلبات،و تعمل بلا نهاية أو حتى يتعطل الخادم (أو يفشل الخادم في البدء) وترجع خطأ دائمًا يكون non-nil.

تخيل موقع تجارة إلكترونية لديه قاعدة بيانات تضع خريطة للأغراض المعروضة للبيع وأسعارها بالدولار، انظر إلى البرنامج أدناه فهو أبسط تطبيق ممكن، فإنه يصنع نموذجًا للحصر بنوع خريطة database والذي إليه نرفق طريقة ServeHTTP حتى ترضي واجهة http.Handler، حيث يقوم المعالج أو المداول handler بالطواف فوق الخريطة وطباعة العناصر.

```go
gopl.io/ch7/http1

func main() {
    db := database{"shoes": 50, "socks": 5}
    log.Fatal(http.ListenAndServe("localhost:8000", db))
}
type dollars float32
func (d dollars) String() string { return fmt.Sprintf("$%.2f", d) }
type database map[string]dollars
func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
    }
}
```

في حال قمنا بتشغيل الخادم:

```
$ go build gopl.io/ch7/http1
$ ./http1 &
```

ثم ربطناه مع برنامج fetch الموضح في القسم 1.5 (أو متصفح إنترنت حسب رغبتك) سنحصل على المخرج التالي:

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://localhost:8000
shoes: $50.00
socks: $5.00
```

حتى الآن يستطيع الخادم فقط عرض كامل مخزونه وسيقوم بذلك عند كل طلب بغض النظر عن الرابط URL، تُعرِّف الخوادم الأكثر واقعية روابط مختلفة متنوعة وكل منها يحدث سوكًا مختلفًا، دعنا نسمي الموجود لدنيا بـ list/ ونضيف واحد آخر باسم price/ حيث يقوم بالإخبار عن سعر كل عنصر محدد كمعامل طلب مثل price?item=socks/ .

```
gopl.io/ch7/http2

func (db database) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    switch req.URL.Path {
    case "/list":
        for item, price := range db {
            fmt.Fprintf(w, "%s: %s\n", item, price)
        }
    case "/price":
        item := req.URL.Query().Get("item")
        price, ok := db[item]
        if !ok {
            w.WriteHeader(http.StatusNotFound) // 404
            fmt.Fprintf(w, "no such item: %q\n", item)
            return
        }
        fmt.Fprintf(w, "%s\n", price)
    default:
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such page: %s\n", req.URL)
    }
}
```

الآن يقرر المداول أي منطق يجب عليه تنفيذه طبقًا لمكون مسار الرابط req.URL.Path، وفي حال لم يستطع المداول تمييز المسار سيقوم بالتبليغ عن خطأ HTTP للعميل من خلال استدعاء w.WriteHeader(http.StatusNotFound) ، ويجب أن يكون ذلك قبل كتابة أي نص إلى w (علما أن http.ResponseWriter عبارة عن واجهة أخرى تزود io.Writer بطرق لإرسال رؤوس استجابة HTTP) وبالمثل يمكننا استخدام دالة المنفعة http.Error:

```
msg := fmt.Sprintf("no such page: %s\n", req.URL)
http.Error(w, msg, http.StatusNotFound) // 404
```

تستدعي price/ طريقة استعلام Query لتحليل معاملات طلب HTTP كخريطة، أو بالأصح كخريطة متعددة من النوع url.Values من الحزمة net/url (راجع القسم 6.2.1)، ومن ثم تجد معامل أول عنصر وتنظر إلى سعره، وفي حال لم تجد العنصر ستبلغ عن خطأ.

يوجد أدناه مثال جلسة مع الخادم الجديد:

```
$ go build gopl.io/ch7/http2
$ go build gopl.io/ch1/fetch
$ ./http2 &
$ ./fetch http://localhost:8000/list
shoes: $50.00
```

```
socks: $5.00
$ ./fetch http://localhost:8000/price?item=socks
$5.00
$ ./fetch http://localhost:8000/price?item=shoes
$50.00
$ ./fetch http://localhost:8000/price?item=hat
no such item: "hat"
$ ./fetch http://localhost:8000/help
no such page: /help
```

من الواضح انه بإمكاننا إضافة المزيد من الحالات إلى ServeHTTP ولكن في التطبيقات الواقعية من المناسب تعريف منطق كل حالة بدالة أو طريقة منفردة، بالإضافة إلى ذلك قد تحتاج الروابط ذات الصلة إلى منطق مشابه، على سبيل المثال الملف الذي يحتوي على عدة صور لديه روابط بالصيغة /images/*.png ، ولهذا السبب تزودنا net/http بـ ServeMux وهو عبارة عن مضاعف إرسال الطلبات request multiplexer وذلك لتسهيل الربط بين الروابط والمداولات، وتقوم ServeMux بتجميع كل الـ http.Handler في http.Handler واحدة. كما نرى أن مختلف الأنواع ترضي نفس الواجهة وهذه الأنواع قابلة للاستبدال، حيث بإمكان خادم الشبكة إيفاد الطلبات إلى أي http.Handler بغض النظر عن النوع المحدد ورائه.

بالنسبة للتطبيقات الأكثر تعقيدًا يمكن تكوين العديد من ServeMuxes للتعامل مع المزيد من متطلبات الإيفاد المعقدة، لا يوجد في لغة جو إطار ويب مرجعي مماثلًا لإطار Ruby Rails أو إطار Python Django، وهذا لا يعني عدم وجود مثل تلك الأطر وإنما قطع البناء في مكتبة جو القياسية قابلة للتعديل بسهولة لدرجة أنه ليس هناك حاجة للأطر، ورغم أنه من المناسب استخدام الأطر في المراحل الأولى للمشروع إلا أنها تضيف تعقيدًا يجعل عملية الصيانة والتحسين صعبة على المدى الطويل.

في البرنامج أدناه قمنا بإنشاء ServeMux واستخدمناه لربط الروابط مع المداولات المحيطة للعماليات /list و/price، والتي قد قسمت إلى طريقتين منفصلتين، ثم استخدمنا ServeMux كمداول رئيس في استدعاء ListenAndServe.

```
gopl.io/ch7/http3
func main() {
    db := database{"shoes": 50, "socks": 5}
    mux := http.NewServeMux()
    mux.Handle("/list", http.HandlerFunc(db.list))
    mux.Handle("/price", http.HandlerFunc(db.price))
    log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type database map[string]dollars

func (db database) list(w http.ResponseWriter, req *http.Request) {
    for item, price := range db {
        fmt.Fprintf(w, "%s: %s\n", item, price)
```

```
    }
}

func (db database) price(w http.ResponseWriter, req *http.Request) {
    item := req.URL.Query().Get("item")
    price, ok := db[item]
    if !ok {
        w.WriteHeader(http.StatusNotFound) // 404
        fmt.Fprintf(w, "no such item: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", price)
}
```

دعونا نركز على الاستدعائيين إلى mux.Handle الذين يسجلان المداولات، في الاستدعاء الأول db.list هو قيمة طريقة والتي هي قيمة من النوع أدناه (راجع القسم 6.4).

```
func(w http.ResponseWriter, req *http.Request)
```

وعندما يستدعى ذلك النوع يستحضر طريقة database.list بقيمة مستقبل db، وبالتالي db.list عبارة عن وظيفة تنفذ السلوكيات الشبيهة بالمداول ولكن بما أنها لا تحتوي على طرق فإنها لا ترضي واجهة http.Handler ولا يمكن تمريرها مباشرة إلى mux.Handle.

التعبير http.HandlerFunc(db.list) هو عبارة عن تحويل وليس استدعاء لدالة، وبما أن http.HandlerFunc يعتبر كنوع فلديه التعاريف التالية:

```
net/http

package http
type HandlerFunc func(w ResponseWriter, r *Request)
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

يوضح HandlerFunc بعض الخصائص الاستثنائية لآلية واجهة جو، إنه نوع دالة لديه طرق ترضي واجهة http.Handler، وسلوكيات طريقته ServeHTTP تستدعي الدالة الرئيسة، وبالتالي تعتبر HandlerFunc مهيئ لقيمة الدالة حتى ترضي الواجهة، في حين أن وظيفة وطريقة الواجهة الوحيدة لديهما نفس التوقيع مما يتيح لبعض الأنواع مثل database إرضاء واجهة http.Handler بطرق مختلفة متعددة، مثل طريقة list وطريقة price وغيرهم.

فمن الشائع تسجيل المداول بهذه الطريقة وبالتالي تحتوي ServeMux على طريقة ملاءمة تسمى HandleFunc، وبالتالي يمكننا تبسيط كود تسجيل المداول كالتالي:

```
gopl.io/ch7/http3a
```

```
mux.HandleFunc("/list", db.list)
mux.HandleFunc("/price", db.price)
```

الكود أعلاه يبين كيف يمكننا بناء برنامج بحيث يحتوي على خادمين اثنين مختلفين لمنفذين مختلفين وتعريف روابط مختلفة والإيفاد لمداولين مختلفين، ويمكننا بناء ServeMux آخر لاستدعاء ListenAndServe، ولكن في معظم البرامج تكتفي بخادم ويب واحد، وأيضًا يفضل تعريف مداولي HTTP في عدة ملفات التطبيق، فمن المزعج إذا كانوا جميعهم مسجلين مباشرة بواجهة التطبيق ServeMux.

وبالتالي لتوفير الراحة للمبرمج تزودنا net/http بحالة عالمية ServeMux تسمى DefaultServeMux وبوظائف على مستوى الحزمة تسمى http.Handle و http.HandleFunc. وفي حال أردنا استخدام ServeMux كمداول رئيسي للخادم ليس علينا تمريره إلى ListenAndServe لأن nil ستقوم بذلك تلقائيًا.

يمكن تبسيط الدالة الرئيسية للخادم كالتالي:

```
gopl.io/ch7/http4
func main() {
    db := database{"shoes": 50, "socks": 5}
    http.HandleFunc("/list", db.list)
    http.HandleFunc("/price", db.price)
    log.Fatal(http.ListenAndServe("localhost:8000", nil))
}
```

أخيرًا من الضروري تذكر الذي ذكرناه في القسم 1.7، وهو أن خادم الويب يستحضر كل مداول في روتين-جو جديد، وبالتالي يجب على المداولين أخذ احتياطاتهم مثل القفل locking عند الوصول لمتغيرات متشعبات أخرى بما في ذلك الطلبات الأخرى لنفس المداول، سنشرح عن التزامن في الفصلين التاليين.

تمرين 7.11: أضف مداولات إضافية حتى يتمكن العميل من إنشاء وقراءة وتحديث وحذف مدخلات قاعدة البيانات، على سبيل المثال الطلب من النموذج /update?item=socks&price=6 سيحدث سعر السلعة في المخزن ويبلغ عن وجود خطأ في حالة لم يجد السلعة أو في حالة كان السعر غير صحيحًا. (تنبيه: هذا التغيير سيسبب في تحديث متغير التزامن).

تمرين 7.12: غير المداول لـ /list ليطبع مخرجاته كجدول HTML أي ليس نصاً، استفيد من حزمة html/template الموضحة في القسم 4.6.

7.8 واجهة الخطأ error

منذ بداية الكتاب كنا نستخدم وننشئ قيم لنوع الخطأ المعلن مسبقًا بدون توضيح ماهيته، في الواقع إنه مجرد نوع واجهة بطريقة واحدة ترجع رسالة خطأ:

```
type error interface {
    Error() string
}
```

أبسط طريقة لإنشاء خطأ هي من خلال استدعاء errors.New مما يرجع خطأ جديد لرسالة خطأ معطاه، طول حزمة الخطأ أربعة أسطر فقط:

```
package errors
func New(text string) error { return &errorString{text} }
type errorString struct { text string }
func (e *errorString) Error() string { return e.text }
```

النوع الرئيسي لـ errorString هو بنية وليست سلسلة وذلك لحماية تمثيلها من التحديثات المقصودة، والسبب وراء أن نوع المؤشر هو errorString* وليس errorString بدون نجمة هو ليرضي واجهة الخطأ لأن كل استدعاء لـ New يحدد حالة خطأ مميزة لا تشبه غيرها. فنحن لا نريد من الخطأ مثل io.EOF أن يتشابه مع خطأ آخر ليعطي نفس الرسالة.

```
fmt.Println(errors.New("EOF") == errors.New("EOF")) // "false"
```

ليس من الشائع استدعاء errors.New بسبب وجود دالة تجميع ملائمة fmt.Errorf تقوم بتنسيق السلسلة النصية أيضًا، استخدمناها عدة مرات في الفصل الخامس.

```
package fmt
import "errors"
func Errorf(format string, args ...interface{}) error {
    return errors.New(Sprintf(format, args...))
}
```

بالرغم من أن errorString* أبسط أنواع الخطأ ولكنها ليست الوحيدة المستخدمة، على سبيل المثال تقدم الحزمة syscall نظام جو منخفض المستوى يسمى واجهة برمجة التطبيقات API، على العديد من المنصات إنها تعرِّف نوعا عدديا Errno يرضي error، وعلى منصات يونيكس طريقة الخطأ Errno تبحث في جداول السلاسل كما هو موضح أدناه:

```
package syscall

type Errno uintptr // operating system error code

 var errors = [...]string{
    1: "operation not permitted", // EPERM
    2: "no such file or directory", // ENOENT
    3: "no such process", // ESRCH
```

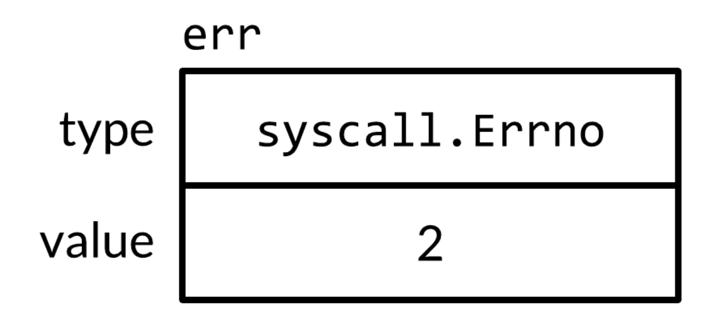
```
	// ...
}

func (e Errno) Error() string {
	if 0 <= int(e) && int(e) < len(errors) {
		return errors[e]
	}
	return fmt.Sprintf("errno %d", e)
}
```

الجملة التالية تنشئ قيمة واجهة تحمل قيمة Errno 2، مما يرضي الشرط POSIX ENOENT:

```
var err error = syscall.Errno(2)
fmt.Println(err.Error())    // "no such file or directory"
fmt.Println(err)            // "no such file or directory"
```

قيمة err موضحة في الصورة 7.6.



صورة 6.7: قيمة الواجهة تحمل العدد الصحيح syscall.Errno

يعتبر Errno تمثيلًا فعالا لنظام يستدعي أخطاء مسحوبة من مجموعة محدودة وترضي واجهة الخطأ القياسية، سنرى المزيد من الأنواع التي ترضي هذه الواجهة في القسم 7.11.

7.9 مثال: مُقيّم التعبير Expression Evaluator

في هذا القسم سنبني مقيم للتعابير الحسابية البسيطة، سنستخدم الواجهة Expr لتمثيل أي تعبير في هذه اللغة، في الوقت الحالي لا تحتاج هذه الواجهة لأي طريقة ولكن سنضيف بعض الطرق لاحقًا.

```
// An Expr is an arithmetic expression.
type Expr interface{}
```

لغة التعابير لدينا تتكون من معطيات أو ثوابت النقطة العائمة floating-point والعمليات الحسابية الثنائية +و - و * و / والعمليات الحسابية الأحادية -x و +x ووظائف تستدعي pow(x,y) و sin(x) و sqrt(x) ومتغيرات مثل x و pi وأقواس وأسبقية المعامل الرياضي القياسية. جميع القيم من النوع float64، ها هنا مجموعة من الأمثلة عن التعابير:

```
sqrt(A / pi)
pow(x, 3) + pow(y, 3)
(F - 32) * 5 / 9
```

تمثل الأنواع الخمسة المحددة أدناه أنواع محددة من التعابير، يمثل Var مرجع إلى المتغير، (سنرى قريبًا لماذا هو مُصدَّرًا)، يمثل (literal) ثابت النقطة العائمة، وتمثل الأنواع الأحادية unary والثنائية binary تعابير العامل بمعامل أو معاملين والذي قد يكون أي نوع من Expr، تمثل call دالة الاستدعاء، وسنحصر حيزها fn ضمن pow أو sin أو sqrt.

```
gopl.io/ch7/eval

// A Var identifies a variable, e.g., x.
type Var string

// A literal is a numeric constant, e.g., 3.141.
type literal float64

// A unary represents a unary operator expression, e.g., -x.
type unary struct {
    op rune // one of '+', '-'
    x  Expr
}

// A binary represents a binary operator expression, e.g., x+y.
type binary struct {
    op   rune // one of '+', '-', '*', '/'
    x, y Expr
}

// A call represents a function call expression, e.g., sin(x).
type call struct {
    fn   string // one of "pow", "sin", "sqrt"
    args []Expr
}
```

لتقييم تعبير يحتوي على متغيرات نحتاج إلى بيئة Env لإنشاء خريطة map تربط أسماء المتغيرات مع القيم:

```
type Env map[Var]float64
```

ونحتاج أيضًا من كل نوع من التعابير أن يعرِّف طريقة Eval لإرجاع قيمة المتغير في البيئة المعنية، وبما أنه يجب على كل تعبير تقديم هذه الطريقة سنضيفه إلى واجهة Expr، تُصدِر الحزمة الأنواع التالية فقط Expr و Env و Var، يمكن للعملاء استخدام المقيم دون الوصول لأنواع التعابير الأخرى.

```
type Expr interface {
    // Eval returns the value of this Expr in the environment env.
    Eval(env Env) float64
}
```

طرق Eval المحددة موضحة أدناه، تقوم طريقة Var ببحث بيئي لاسترجاع الصفر في حال لم يتم تعريف المتغيرات، وتقوم طريقة literal باسترجاع القيمة الحرفية.

```
func (v Var) Eval(env Env) float64 {
    return env[v]
}

func (l literal) Eval(_ Env) float64 {
    return float64(l)
}
```

تقوم طرق Eval للعمليات الأحادية والثنائية بالتقييم المتكرر لمعاملاتها، ثم تطبيق عملية op عليهم، لا نعتبر القسمة على صفر أو ما لانهاية كخطأ لأنهما ينتجان نتيجة حتى ولو كانت لا نهائية، وأخيرًا تقوم طريقة call بتقييم معطيات دالة pow أو sin أو sqrt ومن ثم تستدعي الدالة المسؤولة في حزمة math.

```
func (u unary) Eval(env Env) float64 {
    switch u.op {
    case '+':
        return +u.x.Eval(env)
    case '-':
        return -u.x.Eval(env)
    }
    panic(fmt.Sprintf("unsupported unary operator: %q", u.op))
}

func (b binary) Eval(env Env) float64 {
    switch b.op {
    case '+':
        return b.x.Eval(env) + b.y.Eval(env)
    case '-':
        return b.x.Eval(env) - b.y.Eval(env)
    case '*':
        return b.x.Eval(env) * b.y.Eval(env)
    case '/':
        return b.x.Eval(env) / b.y.Eval(env)
    }
    panic(fmt.Sprintf("unsupported binary operator: %q", b.op))
}

func (c call) Eval(env Env) float64 {
    switch c.fn {
    case "pow":
        return math.Pow(c.args[0].Eval(env), c.args[1].Eval(env))
    case "sin":
        return math.Sin(c.args[0].Eval(env))
    case "sqrt":
        return math.Sqrt(c.args[0].Eval(env))
```

```
    }
    panic(fmt.Sprintf("unsupported function call: %s", c.fn))
}
```

من الممكن أن تفشل العديد من هذه الطرق، على سبيل المثال يمكن للتعبير call أن يأخذ دالة مجهولة أو رقم خاطئ من المعطيات، ومن الممكن أيضًا بناء تعبير أحادي أو ثنائي بعامل خاطئ مثل ! أو > (علمًا أن الدالة Parse المذكورة أدناه لا تفعل ذلك أبدًا. تلك الأخطاء تسبب الهلع لـ Eval، والأخطاء الأخرى مثل تقييم Var غير موجود في البيئة يسبب فقط بإرجاع Eval للنتيجة الخطأ، يمكن الكشف عن جميع تلك الأخطاء من خلال تفقد Expr قبل تقييمه، وهذه هي وظيفة طريقة Check والتي سنشرحها بعد قليل ولكن أولًا دعنا نفحص Eval.

دالة TestEval الموضحة أدناه هي عبارة عن فحص للمقيّم، فهي تستخدم الحزمة testing والتي سنشرحها في الفصل الحادي عشر ولكن ما يجب عليك معرفته الآن هو أن استدعاء t.Errorf يبلغ عن الخطأ. تقوم الدالة بالمرور على جدول المدخلات التي تحدد ثلاثة تعابير والبيئة الخاصة لكل منها، التعبير الأول يحسب نصف قطر دائرة معلوم مساحتها A، والثانية تحسب مجموع تكعيب المتغيرين x و y، وتقوم الثالثة بتحويل درجة الحرارة من فهرنهايت F إلى مئوية.

```
func TestEval(t *testing.T) {
    tests := []struct {
        expr string
        env  Env
        want string
    }{
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 12, "y": 1}, "1729"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
        {"5 / 9 * (F - 32)", Env{"F": 32}, "0"},
        {"5 / 9 * (F - 32)", Env{"F": 212}, "100"},
        //!-Eval
        // additional tests that don't appear in the book
        {"-1 + -x", Env{"x": 1}, "-2"},
        {"-1 - x", Env{"x": 1}, "-2"},
        //!+Eval
    }
    var prevExpr string
    for _, test := range tests {
        // Print expr only when it changes.
        if test.expr != prevExpr {
            fmt.Printf("\n%s\n", test.expr)
            prevExpr = test.expr
        }
        expr, err := Parse(test.expr)
        if err != nil {
            t.Error(err) // parse error
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        fmt.Printf("\t%v => %s\n", test.env, got)
        if got != test.want {
            t.Errorf("%s.Eval() in %v = %q, want %q\n",
                test.expr, test.env, got, test.want)
```

```
        }
    }
}
```

عند كل إدخال في الجدول يقوم الفحص بتحليل التعبير وتقييمه في البيئة ويطبع نتائجه، لا يوجد مساحة كافية لعرض دالة Parse هنا ولكن ستجدها عند تحميل الحزمة باستخدام go get.

يقوم الأمر go test بتفعيل فحص الحزمة (راجع القسم 11.1):

```
$ go test -v gopl.io/ch7/eval
```

يمكننا العَلَم v- من رؤية المخرج المطبوع للفحص ، وعادة ما يتجاهل للفحص الناجح مثل هذا، ها هو مخرج جملة الفحص fmt.Printf:

```
sqrt(A / pi)
    map[A:87616 pi:3.141592653589793] => 167

pow(x, 3) + pow(y, 3)
    map[x:12 y:1] => 1729
    map[x:9 y:10] => 1729

5 / 9 * (F - 32)
    map[F:-40] => -40
    map[F:32] => 0
    map[F:212] => 100
```

لحسن الحظ صيغت جميع المداخل جيدًا لحد الآن، ولكن حظنا السعيد لن يدوم، لأنه حتى في اللغات المُفسرة من الشائع تفقد بناء الجملة لإيجاد الأخطاء الثابتة وهذا خطأ يمكن إيجاده دون تشغيل البرنامج، فيمكننا إيجاد الأخطاء بشكل أسرع من خلال فصل الاختبارات الثابتة عن الاختبارات الديناميكية، وتشغيل الاختبارات مرة واحدة بدلا عن كل مرة يتم فيها تقييم التعبير.

دعنا نضيف طريقة أخرى للواجهة Expr، فتقوم طريقة Check بتفقد الأخطاء الثابتة في بناء جملة التعبير، سنشرح معامل vars الخاصة به بعد قليل.

```go
type Expr interface {
    Eval(env Env) float64
    // Check reports errors in this Expr and adds its Vars to the set.
    Check(vars map[Var]bool) error
}
```

طريقة Check المحددة موضحة أدناه، لا يمكن فشل فحص literal و Var، وبالتالي طرق Check لهذه الأنواع ترجع nil. تقوم طرق العمليات الآحادية والثنائية بفحص صلاحية العامل ثم تفحص المعطيات، وعلى مثيل ذلك طريقة call تقوم بفحص إذا كانت الدالة معلومة ولديها العدد الصحيح من المعاملات ثم تفحص كل معامل.

```
func (v Var) Check(vars map[Var]bool) error {
    vars[v] = true
    return nil
}

func (literal) Check(vars map[Var]bool) error {
    return nil
}

func (u unary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-", u.op) {
        return fmt.Errorf("unexpected unary op %q", u.op)
    }
    return u.x.Check(vars)
}

func (b binary) Check(vars map[Var]bool) error {
    if !strings.ContainsRune("+-*/", b.op) {
        return fmt.Errorf("unexpected binary op %q", b.op)
    }
    if err := b.x.Check(vars); err != nil {
        return err
    }
    return b.y.Check(vars)
}

func (c call) Check(vars map[Var]bool) error {
    arity, ok := numParams[c.fn]
    if !ok {
        return fmt.Errorf("unknown function %q", c.fn)
    }
    if len(c.args) != arity {
        return fmt.Errorf("call to %s has %d args, want %d",
            c.fn, len(c.args), arity)
    }
    for _, arg := range c.args {
        if err := arg.Check(vars); err != nil {
            return err
        }
    }
    return nil
}

var numParams = map[string]int{"pow": 2, "sin": 1, "sqrt": 1}
```

لقد أدرجنا قائمة بالمدخلات الخاطئة المختارة والخطأ الذي تسببه في مجموعتين، تبلغ الدالة Parse (غير ظاهرة) عن أخطاء بناء الجملة وتبلغ الدالة Check عن الأخطاء الدلالية.

```
x % 2       unexpected  '%'
math.Pi     unexpected  '.'
!true       unexpected  '!'
"hello"     unexpected  '"'

log(10)     unknown function "log"
sqrt(1,2)   call to sqrt has 2 args, want 1
```

معاملات Check هي عبارة عن مجموعة من المتغيرات وتراكم مجموعة أسماء المتغيرات الموجودة ضمن التعبير، يجب تواجد كل من هذه المتغيرات ضمن البيئة للتقييم لكي تنجح، هذه المجموعة عبارة عن نتائج استدعاء Check ولكن بما أن الطريقة متكررة فمن الأفضل تجميع مجموعة كمعاملات، يجب على العميل مجموعة فارغة في الاستدعاء الابتدائي.

قمنا في القسم 3.2 برسم الدالة f(x,y) بحيث كانت ثابته على وقت المترجم، ولكن بما أنه الآن يمكننا تحليل وفحص وتقييم التعابير في السلسلة النصية فيمكننا بناء تطبيق ويب يستقبل التعبير وقت التشغيل من العميل ورسم سطح الدالة، يمكننا استخدام مجموعة vars لفحص أن التعبير عبارة عن دالة متغيرين فقط x و y، أو بالأصح ثلاثة متغيرات كوننا سنستخدم المتغير r للدلالة على نصف القطر، وسنستخدم طريقة Check لرفض جميع التعابير الخاطئة قبل بدء التقييم وبالتالي لن نكرر هذه الاختبارات خلال 40 ألف فحص (100*100) خلية وكل منهم لها أربع زوايا) للدالة التي تتبع.

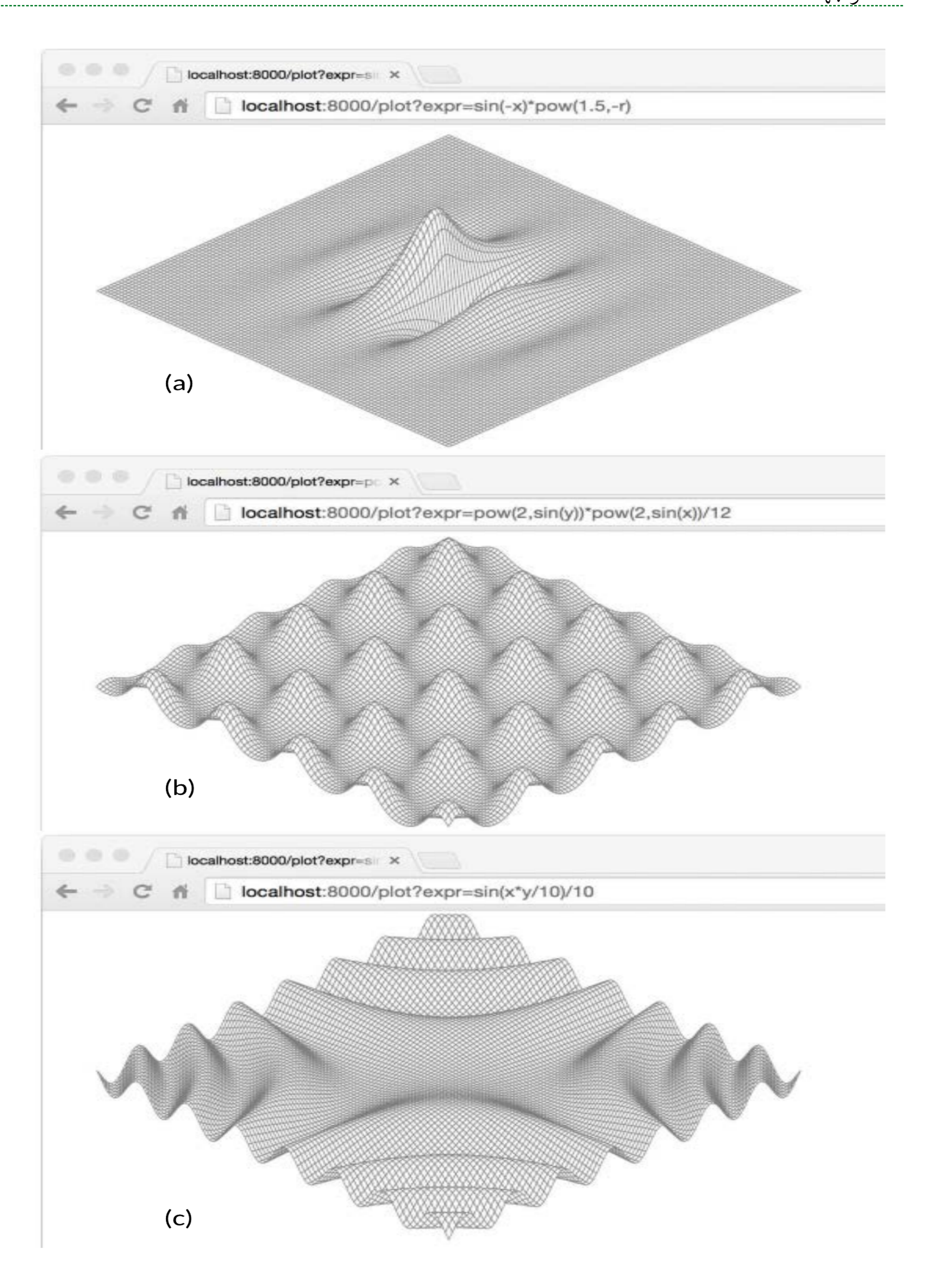
تجمع الدالة parseAndCheck بين خطوات الفحص والتحليل:

```go
gopl.io/ch7/surface

func parseAndCheck(s string) (eval.Expr, error) {
    if s == "" {
        return nil, fmt.Errorf("empty expression")
    }
    expr, err := eval.Parse(s)
    if err != nil {
        return nil, err
    }
    vars := make(map[eval.Var]bool)
    if err := expr.Check(vars); err != nil {
        return nil, err
    }
    for v := range vars {
        if v != "x" && v != "y" && v != "r" {
            return nil, fmt.Errorf("undefined variable: %s", v)
        }
    }
    return expr, nil
}
```

ولجعله تطبيق ويب سنستخدم دالة plot أدناه علمًا أنه لديها نفس توقيع http.HandlerFunc

```go
func plot(w http.ResponseWriter, r *http.Request) {
	r.ParseForm()
	expr, err := parseAndCheck(r.Form.Get("expr"))
	if err != nil {
		http.Error(w, "bad expr: "+err.Error(), http.StatusBadRequest)
		return
	}
	w.Header().Set("Content-Type", "image/svg+xml")
	surface(w, func(x, y float64) float64 {
		r := math.Hypot(x, y) // distance from (0,0)
		return expr.Eval(eval.Env{"x": x, "y": y, "r": r})
	})
}
```



صورة 7.7: سطح الوظائف الثلاث: (أ) sin(-x)*pow(1.5,-r)

(ب) pow(2,sin(y))*pow(2,sin(x))/12 (ج) sin(x*y/10)/10

تقوم دالة الرسم plot بتحليل وفحص التعبير المحدد في طلب HTTP وتستخدمه لإنشاء دالة مجهولة من متغيرين، لدى الدالة المجهولة نفس توقيع الدالة الثابتة f من برنامج رسم الأسطح الأصلي، ولكنها تقيّم التعبير الذي يقدمه المستخدم، وتحدد البيئة x و y ونصف القطر r. أخيرًا تقوم plot باستدعاء surface وهي مجرد دالة رئيسية من gopl.io/ch3/surface ولكن عدلت لتأخذ الدالة إلى plot وتأخذ المخرج إلى io.Writer كمعاملات بدلًا من استخدام الدالة الثابتة f و os.Stdout. يوضح الشكل 7.7 ثلاث أسطح ناشئة من البرنامج.

تمرين 7.13: أضف طريقة String إلى Expr لطباعة شجرة بناء الجملة وافحص النتائج بحيث عند تحليلها مرة أخرى تعطي شجرة مرادفة.

تمرين 7.14: عرف نوعا محددا جديدا يرضي واجهة Expr وقدّم عملية جديدة مثل حساب القيمة الدنيا لمعطياتها، وبما أن دالة Parse لا تنشئ حالات لهذا النوع الجديد، وبالتالي لتستخدمه يجب عليك بناء شجرة بناء الجملة مباشرة أو قم بإطالة المحلل.

تمرين 7.15: اكتب برنامجا يقرأ تعبيرًا واحدًا من المدخل القياسي وطالب المستخدم أن يزودك بقيم لأي متغيرات ثم قيِّم التعبير في البيئة الناتجة، تعامل مع الأخطاء بذكاء.

تمرين 7.16: قم بكتابة برنامج آلة حاسبة كتطبيق وب.

7.10 تأكيد النوع Type Assertions

تأكيد النوع هو عبارة عن عملية تُطبَّق على قيمة الواجهة، فهي تشبه من الناحية النحوية x.(T) ، حيث أن x هو تعبير نوع الواجهة و T هي النوع وتسمى بالنوع المؤكد (asserted type)، يقوم تأكيد النوع بفحص ما إذا كان النوع الديناميكي لمعطياته يناسب النوع المؤكد.

يوجد لدينا احتمالان، الأول هو أن يكون النوع المؤكد T عبارة عن نوع محدد، وبالتالي يقوم تأكيد النوع بفحص ما إذا كان النوع الديناميكي لـ x مطابقًا لـ T، وفي حال نجاح الفحص فإن نتائج توكيد النوع هي النوع الديناميكي لـ x، والذي نوعه بالطبع T، أي بمعنى آخر تأكيد النوع لنوع محدد يستخلص القيمة المحددة من معطياته، ولكن في حال فشل الفحص فسيحصل خطأ (هلع) في العملية، على سبيل المثال:

```
var w io.Writer
w = os.Stdout
f := w.(*os.File)      // success: f == os.Stdout
c := w.(*bytes.Buffer) // panic: interface holds *os.File, not *bytes.Buffer
```

والاحتمالية الثانية، أن نوع T هو نوع واجهة، وبالتالي سيقوم تأكيد النوع بفحص ما إذا كان النوع الديناميكي لـ x يرضي T، في حال نجاح الفحص فلن يتم استخلاص القيمة الديناميكية، وبالتالي ستبقى النتائج عبارة عن قيمة الواجهة بنفس مكونات النوع والقيمة، ولكن لدى النتيجة نوع الواجهة T، أي بمعنى آخر تأكيد النوع لنوع واجهة يغير نوع التعبير صانعا مجموعة مختلفة (وغالبًا كبيرة) من إمكانيات الوصول للطرق، ولكنها تحتفظ بالقيمة والنوع الديناميكي داخل قيمة الواجهة.

بعد أول تأكيد للنوع الموضح أدناه، كلا w و rw ستحملان os.Stdout وبالتالي كل منهما لديه نوع ديناميكي *os.File ولكن w والتي هي io.Writer تكشف طريقة Write للملف بينما تكشف rw طريقة Read للملف.

```
var w io.Writer
w = os.Stdout
rw := w.(io.ReadWriter) // success: *os.File has both Read and Write
w = new(ByteCounter)
rw = w.(io.ReadWriter) // panic: *ByteCounter has no Read method
```

بغض النظر عن النوع الذي تم توكيده فإن تأكيد النوع سوف يفشل طالما المعطيات كانت قيمة واجهة nil، ونادرًا ما يكون هناك الحاجة لتوكيد النوع لنوع واجهة أقل تقيدًا (طرق قليلة)، حيث يكون سلوكها كإسناد ولكن في حالة nil.

```
w = rw                 // io.ReadWriter is assignable to io.Writer
w = rw.(io.Writer)   // fails only if rw == nil
```

غالبًا نكون غير موقنين من النوع الديناميكي لقيمة الواجهة ونود فحص ما إذا كان نوعًا معينًا، في حال ظهر تأكيد النوع في معامل يكون فيه نتيجتين متوقعتين كما هو موجود في الإعلان التالي، فالعملية لا تفشل ولكن تُرجع نتيجة ثانية إضافية، قيمة منطقية تشير إلى النجاح:

```
var w io.Writer = os.Stdout
f, ok := w.(*os.File)      // success: ok, f == os.Stdout
b, ok := w.(*bytes.Buffer) // failure: !ok, b == nil
```

تُعين النتيجة الثانية لمتغير اسمه ok، ففي حال فشل العملية سيكون ok يحمل قيمة false، والنتيجة الأولى مساوية لقيمة الصفر للنوع المؤكد والذي هو في هذا المثال صفر *bytes.Buffer

غالبًا تستخدم نتيجة ok مباشرة لتحديد ما يجب فعله تاليًا، تساعد الشكل الممتد من جملة if بتقليص الحجم:

```
if f, ok := w.(*os.File); ok {
    // ...use f...
}
```

وعندما يكون معطى توكيد النوع عبارة عن متغير، فبدلًا من تأليف اسم جديد للمتغير المحلي الجديد سترى بعض الأحيان الاسم الجديد قد أعيد استخدامه مظللًا للأصلي كالتالي:

```
if w, ok := w.(*os.File); ok {
     // ...use w...
}
```

7.11 تمييز الأخطاء في تأكيد النوع

اعتبر أن مجموعة من الأخطاء استرجعت بواسطة عمليات الملف في حزمة os، فسوف يفشل (I/O) لعدة أسباب ولكن هناك ثلاثة أنواع من الفشل يتعامل معها بشكل مختلف وهي: ملف موجود مسبقًا (لعمليات الإنشاء)، وملف غير موجود (لعمليات القراءة) والأذن مرفوض. تقدم حزمة os ثلاث وظائف مساعدة لتصنيف الفشل المكتشف حسب قيمة خطأ معطاه:

```
package os
func IsExist(err error) bool
func IsNotExist(err error) bool
func IsPermission(err error) bool
```

فالتنفيذ الساذج لأي من الجمل الثلاث قد يفحص احتواء رسالة الخطأ على سلسلة فرعية معينة.

```
func IsNotExist(err error) bool {
     // NOTE: not robust!
     return strings.Contains(err.Error(), "file does not exist")
}
```

ولكن بما أن المنطق للتعامل مع أخطاء I/O قد يختلف من منصة إلى أخرى فهذا النهج ليس موثوقًا بما فيه الكفاية وقد يبلغ عن بعض حالات الفشل بعدة رسائل خطأ مختلفة، قد يكون مفيدًا فحص السلاسل الفرعية لرسائل الخطأ للتأكد من أن الوظائف تفشل في الأسلوب المتوقع ولكنها غير كافية للشفرة المؤهلة لمنصة الإنتاج.

النهج الأكثر موثوقية هو تمثيل قيم الخطأ المبنية بواسطة نوع مخصص، فتعرف الحزمة os نوعًا اسمه PathError ليصف حالات الفشل التي تشمل عمليات على مسار الملف مثل Open أو Delete، وهناك بديل يسمى LinkError يقوم بوصف حالات الفشل التي تشمل عمليات على مسارين ملف مثل Symlink و Rename، وها هنا os.PathError:

```
package os

// PathError records an error and the operation and file path that caused it.
type PathError struct {
   Op
   string
   Path string
   Err error
}
```

```
func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

يكون معظم العملاء واضحين في PathError ويتعاملون مع الأخطاء بطريقة موحدة من خلال استدعاء طرق Error الخاصة بهم، بالرغم من ذلك تقوم طريقة الخطأ PathError بتشكيل رسالة من خلال ربط الحقول، وتحفظ بنية PathError المكونات الأساسية للخطأ، فيستطيع العميل التمييز بين أنواع الفشل باستخدام توكيد النوع ليكشف النوع المحدد للخطأ حيث أن النوع المحدد يقدم تفاصيل أكثر من السلسلة البسيطة.

```
_, err := os.Open("/no/such/file")
fmt.Println(err) // "open /no/such/file: No such file or directory"
fmt.Printf("%#v\n", err)
// Output:
// &os.PathError{Op:"open", Path:"/no/such/file", Err:0x2}
```

فذلك كان مبدأ عمل وظائف المساعدة الثلاث، على سبيل المثال تبلغ IsNotExist المبينة أدناه ما إذا الخطأ مساويًا لـ syscall.ENOENT (راجع القسم 7.8) أو للخطأ المتميز os.ErrNotExist (أنظر إلى io.EOF في القسم 5.4.2) أو هو

PathError* حيث خطأه الأساسي هو واحد من ذلك الاثنين.

```
import (
"errors"
"syscall"
)

var ErrNotExist = errors.New("file does not exist")

// IsNotExist returns a boolean indicating whether the error is known to
// report that a file or directory does not exist. It is satisfied by
// ErrNotExist as well as some syscall errors.
func IsNotExist(err error) bool {
    if pe, ok := err.(*PathError); ok {
            err = pe.Err
    }
    return err == syscall.ENOENT || err == ErrNotExist
}
```

وها هي أثناء الفعالية:

```
_, err := os.Open("/no/such/file")
fmt.Println(os.IsNotExist(err)) // "true"
```

وبالطبع ستخسر بنية PathError إذا اتحدت رسالة الخطأ مع سلسلة أكبر، مثل استدعاء fmt.Errorf، يجب تمييز الخطأ مباشرة بعد العمليات الفاشلة وقبل إرسال الخطأ إلى المستدعي.

7.12 سلوكيات الاستعلام مع تأكيد نوع الواجهة

يتشابه المنطق أدناه مع الجزء خادم الويب net/http المسؤول عن كتابة حقول عنوان HT TP مثل "Content-type: text/html"، يمثل io.Writer w استجابة HT TP، فتكتب البايتات المكتوبة له تُرسل إلى متصفح الويب لشخص آخر.

```
func writeHeader(w io.Writer, contentType string) error {
	if _, err := w.Write([]byte("Content-Type: ")); err != nil {
		return err
	}
	if _, err := w.Write([]byte(contentType)); err != nil {
		return err
	}
	// ...
}
```

وبما أن طريقة Write تتطلب شريحة بايت، والقيمة التي نود كتابتها هي سلسلة، فلا بد من إجراء التحويل []byte(...)، فهذا التحويل يخصص مواقع الذاكرة ويصنع نسخة ولكن يستغنى عن هذه النسخة بعد كتابة السلسة مباشرة، دعنا نفترض أن هذا جزء أساسي لخادم ويب وأن قياس الأداء الخاص بنا كشفَ أن تخصيص هذه الذاكرة يسبب في إبطاء هذا الجزء ، فهل يمكننا الاستغناء عن تخصيص الذاكرة؟

تخبرنا واجهة io.Writer حقيقة واحدة عن النوع المحدد الذي يمثل w وهو أن البايتات يمكن كتابتها فيه، فلو تعمقنا في البحث في حزمة net/http سنجد أن النوع الديناميكي الذي يمثل w في هذا البرنامج لديه طريقة WriteString مما يتيح إمكانية كتابة السلاسل فيه مما يتيح تجنب تخصيص نسخة مؤقتة. (إنها تبدو كمغامرة في الظلام كون احتمالية نجاحها ضئيلة ولكن هناك العديد من الأنواع المهمة التي ترضي io.Writer ولديها طريقة WriteString بما في ذلك *byte s.Buffer و*os.File و*bufio.Writer).

لا يمكننا افتراض أن io.Writer w عشوائي لديه طريقة WriteString ولكن يمكننا تعريف واجهة جديدة لديها تلك الطريقة ونستخدم تأكيد النوع لفحص ما إذا النوع الديناميكي لـ w يرضي الواجهة الجديدة.

```
// writeString writes s to w.
// If w has a WriteString method, it is invoked instead of w.Write.
func writeString(w io.Writer, s string) (n int, err error) {
	type stringWriter interface {
		WriteString(string) (n int, err error)
	}
	if sw, ok := w.(stringWriter); ok {
		return sw.WriteString(s) // avoid a copy
	}
	return w.Write([]byte(s)) // allocate temporary copy
}
```

```
func writeHeader(w io.Writer, contentType string) error {
	if _, err := writeString(w, "Content-Type: "); err != nil {
		return err
	}
	if _, err := writeString(w, contentType); err != nil {
		return err
	}
	// ...
}
```

ولتجنب تكرار نفس الخطوات قمنا بنقل الفحص إلى دالة المنفعة writeString ولكنه من المفيد جدا معرفة أن المكتبة القياسية تقدمه كـ io.WriteString، إنها الطريقة المفضلة لكتابة سلسلة إلى io.Writer.

المثير للفضول في هذا المثال هو عدم وجود أي واجهة قياسية تُعرّف طريقة WriteString وتحدد سلوكياتها اللازمة، بالإضافة إلى ذلك يوجد النوع المحدد بواسطة طرقه سواء كان يرضي أو لا يرضي واجهة stringWriter، فطرقه كافية لتحديده دون الحاجة لإعلان العلاقة بينه وبين نوع الواجهة، والمغزى من ذلك هو أن التقنيات أعلاه تعتمد على افتراض أن إذا كان النوع يلبي الواجهة أدناه فيجب أن يكون لـ WriteString(s) نفس تأثير Write([]byte(s)).

```
interface {
	io.Writer
	WriteString(s string) (n int, err error)
}
```

بالرغم من أن io.WriteString توثق افتراضها، إلا أن القليل من الوظائف التي تستدعيها توثق أنها أيضًا لها نفس الافتراض، ويُعتبر تعريف طريقة لنوع معين كموافقة ضمنية لعقد سلوكي محدد. المستخدمون الجدد للغة جو وخصوصًا هؤلاء الذين اعتادوا على اللغات ذات الأنواع الصريحة قد يستصعبون قلة وجود البنية الصريحة، ولكن بعد الممارسة ستسهل الأمور عليهم. غالبًا لا ترضي الافتراضات أنواع الواجهات باستثناء الواجهة الفارغة interface{}.

تستخدم الدالة writeString أعلاه تأكيد النوع لمعرفة ما إذا قيمة نوع الواجهة العامة قد ترضي نوعا أكثر تحديدًا، وفي حال كانت ترضيه فإنها تستخدم سلوكيات الواجهة المحددة، يمكن الاستفادة من هذه الآلية سواء كانت الواجهة المستعلم عنها قياسية مثل io.ReadWriter أو معرفة بواسطة المستخدم مثل stringWriter.

ومن خلال هذه الآلية تميز fmt.Fprintf القيم التي ترضي error أو fmt.Stringer من بين جميع القيم، توجد خطوة ضمن fmt.Fprintf تحول معامل واحد إلى سلسلة، كما هو موضح أدناه:

```
package fmt

func formatOneValue(x interface{}) string {
	if err, ok := x.(error); ok {
```

```
                return err.Error()
        }
        if str, ok := x.(Stringer); ok {
                return str.String()
        }
        // ...all other types...
}
```

إذا كانت x ترضي أي من الواجهتين فذلك يحدد تنسيق النوع، وفي حال لم يرضي أي من الواجهتين فالحالة الافتراضية تتولى أمر باقي جميع الأنواع بشكل شبه رسمي من خلال الانعكاس reflection، سنتعلم عن ذلك في الفصل الثاني عشر.

نكرر مرة أخرى، هذا يضع افتراضية أن أي نوع له طريقة String يرضي العقد السلوكي لـ fmt.Stringer وذلك لإرجاع السلسة لتكون مناسبة للطباعة.

7.13 مُبَدِلات النوع

تُستخدم الواجهات ضمن أسلوبين متميزين، في الأسلوب الأول تتجسد الواجهة بـ io.Reader و io.Writer و fmt.Stringer و sort.Interface و http.Handler و error، وتعبر طرق الواجهة عن تشابهات الأنواع المحددة التي ترضي الواجهة ولكن تخفي تفاصيل التمثيل والعمليات الجوهرية لتلك الأنواع المحددة. التركيز هنا على الطرق ليس الأنواع المحددة.

الأسلوب الثاني يستغل إمكانية قيمة الواجهة على حمل قيم مختلف الأنواع المحددة وتعتبر الواجهة كموحد لتلك الأنواع، يستخدم تأكيد النوع للتمييز بين تلك الأنواع بشكل ديناميكي وتعامل كل حالة بشكل مختلف، في هذا الأسلوب يكون التركيز على الأنواع المحددة التي ترضي الواجهة ليس على طرق الواجهة، ولا يوجد إخفاء للمعلومات. سنوصف الواجهات التي تستخدم هذا الأسلوب بالاتحادات المميزة.

في حال سبق لك التعامل مع لغات البرمجة كائنية التوجه فقد تكون على دراية أن هذين الأسلوبين يسميان بـ(تعدد أشكال النوع الفرعي) و(تعدد أشكال مخصص)، ليس من الضرورة معرفة هذين المصطلحين ولكن كن على دراية أن الأمثلة في هذا الفصل ستكون على الأسلوب الثاني.

تمكننا واجهة برمجة التطبيقات جو أن نستعلم قاعدة بيانات SQL ، مثلما هو في بقية اللغات، دعنا نفصل بوضوح بين الأجزاء الثابتة للاستعلام وأجزاء المتغيرات، على سبيل المثال قد يبدو العميل كالتالي:

```
import "database/sql"
func listTracks(db sql.DB, artist string, minYear, maxYear int) {
    result, err := db.Exec(
```

```
                "SELECT * FROM tracks WHERE artist = ? AND ? <= year AND year
<= ?",
                artist, minYear, maxYear)
    // ...
}
```

تقوم طريقة Exec باستبدال كل '?' في سلسلة الاستعلام بلغة استعلام هيكلية literal لتدل على قيمة المعطيات المحيطة والتي قد تكون جملة منطقية أو رقم أو سلسلة أو nil، بناء الاستعلامات بهذه الطريقة يساعد على تجنب الهجمات المحقونة في لغة الاستعلام الهيكلية، والتي من خلالها يسيطر العدو على الاستعلامات من خلال استغلال الاقتباسات غير المناسبة للبيانات المدخلة، قد نجد ضمن Exec وظيفة مثل الموضحة أدناه بحيث تحول قيمة كل معطى إلى ترميز حرفي للغة SQL.

```go
func sqlQuote(x interface{}) string {
    if x == nil {
        return "NULL"
    } else if _, ok := x.(int); ok {
        return fmt.Sprintf("%d", x)
    } else if _, ok := x.(uint); ok {
        return fmt.Sprintf("%d", x)
    } else if b, ok := x.(bool); ok {
        if b {
            return "TRUE"
        }
        return "FALSE"
    } else if s, ok := x.(string); ok {
        return sqlQuoteString(s) // (not shown)
    } else {
        panic(fmt.Sprintf("unexpected type %T: %v", x, x))
    }
}
```

تقوم جملة التبديل switch بتبسيط سلسلة if-else والتي تقوم بسلسلة من اختبارات تساوي القيم، وكذلك تقوم جملة تحويل النوع type switch بتبسيط سلسلة if-else من تأكيدات النوع.

تتشابه جملة مبدل النوع مع جملة التبديل switch حيث معاملها x.(type) (هذا فعليًا النوع الأساسي type) ولدى كل حالة نوع واحد أو أكثر، يتيح مبدل النوع التفرع متعدد الطرق طبقًا للنوع الديناميكي لقيمة الواجهة، في حالة الصفر nil تتوافق إذا كانت x == nil والحالة الافتراضية تتوافق إذا لم تتوافق أي حالة أخرى. إن مبدل النوع sqlQuote قد يملك هذه الحالات:

```go
switch x.(type) {
case nil:       // ...
case int, uint: // ...
case bool:      // ...
case string:    // ...
default:  // ...
```

```
}
```

كما هو الحال مع جملة switch العادية (راجع القسم 1.8) تؤخذ الحالات بالترتيب وعند إيجاد التوافق ينفذ جسم الحالات (النص البرمجي للحالات)، تظهر أهمية ترتيب الحالات عندما يكون واحد أو أكثر من أنواع الحالة عبارة عن واجهات، وبما أنه هناك احتمالية توافق حالتين فموقع الحالة القياسية default بالنسبة للحالات الأخرى غير مهم، لا يتاح استخدام fallthrough.

لاحظ أن في الدالة الأصلية يحتاج منطق الحالات bool و string إلى إمكانية الوصول إلى القيمة المستخلصة من تأكيد النوع، وبما أن هذا الأمر نموذجيًا فلدى جملة مبدل النوع شكل ممتد أطول من ذلك الذي يربط القيمة المستخلصة مع المتغير الجديد ضمن كل حالة:

```
switch x := x.(type) { /* ... */ }
```

قمنا بتسمية المتغيرات الجديدة بـ x أيضًا، علمًا أنه يمكنك إعادة استخدام أسماء المتغيرات مع تأكيدات النوع، وعلى مثيل جملة التبديل switch ، يقوم مبدل النوع بإنشاء كتلة لغوية ضمنيًا، وبالتالي لا يتعارض إعلان المتغير الجديد x مع المتغير x من الكتلة الخارجية. كل حالة تنشئ كتلة لغوية منفصلة بشكل ضمني.

عند إعادة كتابة sqlQuote لاستخدام الشكل الممتد من مبدل النوع يجعل البرنامج أوضح:

```
func sqlQuote(x interface{}) string {
    switch x := x.(type) {
    case nil:
        return "NULL"
    case int, uint:
        return fmt.Sprintf("%d", x) // x has type interface{} here.
    case bool:
        if x {
            return "TRUE"
        }
        return "FALSE"
    case string:
        return sqlQuoteString(x) // (not shown)
    default:
        panic(fmt.Sprintf("unexpected type %T: %v", x, x))
    }
}
```

في هذه النسخة لدى المتغير x ضمن كل حالة نوع نفس نوع الحالة، على سبيل المثال يكون x من نوع bool ضمن حالة bool و string ضمن حالة string. في جميع الحالات الأخرى يكون x من نوع الواجهة لمعامل switch وهذا النوع في هذا المثال هو {}interface. عندما يتطلب نفس الإجراء للحالات المتعددة مثل int و uint يقوم النوع switch بتسهيل الدمج بينهما.

بالرغم من أن sqlQuote تقبل معطى أي نوع إلا أن الدالة تعمل حتى النهاية فقط عندما يتوافق نوع المعطى مع واحدة من الحالات في النوع switch، وإلا سيحدث خطأ برسالة نوع غير متوقع ‘‘unexpected type’’، وبالرغم من أن نوع x هو {}interface إلا أننا نعتبره اتحاد مميز لـ int و uint و bool و string و nil.

7.14 مثال: فك تشفير XML المبنية على الرمز المميز

شرحنا في القسم 4.5 كيفية فك تشفير مستندات جافا سكريبت JSON إلى بنيات بيانات جو باستخدام الدالتين Marshal و Unmarshal من حزمة encoding/json. تزودنا حزمة encoding/xml بواجهة برمجة تطبيقات مشابهة، يكون هذا النهج مناسباً عندما نريد بناء تمثيلًا لشجرة المستند ولكنه ليس ضروريًا للكثير من البرامج. وتقدم الحزمة encoding/xml أيضًا واجهة برمجة تطبيقات مبنية على الرمز المميز token لفك تشفير لغة التوصيف الموسعة XML. في أسلوب الرمز المميز يقوم المحلل بالاستفادة من المداخل لإنتاج مجموعة من الرموز المميزة وبشكل أساسي من أربع أنواع وهي StartElement و EndElement و CharData و Comment. كل منها يكون نوعا محددا ضمن حزمة encoding/xml، وكل منها يستدعي (xml.Decoder.Token*) ويرجع رمزًا مميزًا.

الأجزاء ذات الصلة لواجهة برمجة التطبيقات موضحة أدناه:

```
encoding/xml

package xml

type Name struct {
    Local string // e.g., "Title" or "id"
}

type Attr struct { // e.g., name="value"
    Name Name
    Value string
}

// A Token includes StartElement, EndElement, CharData,
// and Comment, plus a few esoteric types (not shown).
type Token interface{}
type StartElement struct { // e.g., <name>
    Name Name
    Attr []Attr
}
type EndElement struct { Name Name } // e.g., </name>
type CharData []byte                 // e.g., <p>CharData</p>
type Comment []byte                  // e.g., <!-- Comment →
```

```
type Decoder struct{ /* ... */ }
func NewDecoder(io.Reader) *Decoder
func (*Decoder) Token() (Token, error) // returns next Token in sequence
```

لا تحتوي حزمة Token على أي طريقة وتعتبر أيضا مثالًا على الاتحاد المميز، والغاية من استخدام واجهات تقليدية مثل io.Reader هو لإخفاء تفاصيل الأنواع المحددة التي ترضي الواجهة وبالتالي تتمكن من إنشاء تطبيقات جديدة، بحيث يعامل كل نوع محدد بشكل رسمي. وعلى النقيض من ذلك تكون مجموعة الأنواع الثابتة التي ترضي الاتحاد المميز ثابتة حسب التصميم وظاهرة ليست مخفية، لدى أنواع الاتحاد المميز عدة طرق، وتُوصَف الوظائف التي تعمل عليهم كمجموعة من الحالات باستخدام مُبدل النوع ولكن بمنطق مختلف لكل حالة.

يقوم البرنامج xmlselect أدناه باستخلاص وطباعة النص الموجودة أسفل العناصر المحددة في شجرة مستند XML، ومن خلال استخدام واجهة برمجة التطبيقات سيقوم البرنامج بعمله من خلال المرور مرة واحدة على المدخل دون الحاجة لتجسيد الشجرة.

```
gopl.io/ch7/xmlselect

// Xmlselect prints the text of selected elements of an XML document.
package main

import (
    "encoding/xml"
    "fmt"
    "io"
    "os"
    "strings"
)

func main() {
    dec := xml.NewDecoder(os.Stdin)
    var stack []string // stack of element names
    for {
        tok, err := dec.Token()
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Fprintf(os.Stderr, "xmlselect: %v\n", err)
            os.Exit(1)
        }
        switch tok := tok.(type) {
        case xml.StartElement:
            stack = append(stack, tok.Name.Local) // push
        case xml.EndElement:
            stack = stack[:len(stack)-1] // pop
        case xml.CharData:
            if containsAll(stack, os.Args[1:]) {
                fmt.Printf("%s: %s\n", strings.Join(stack, " "), tok)
            }
        }
    }
```

```
}

// containsAll reports whether x contains the elements of y, in order.
func containsAll(x, y []string) bool {
    for len(y) <= len(x) {
        if len(y) == 0 {
            return true
        }
        if x[0] == y[0] {
            y = y[1:]
        }
        x = x[1:]
    }
    return false
}
```

في كل مرة تتلاقى حلقة main مع StartElement فإنها تكدّس اسم العنصر في الرصّة (الكومة) وتسترجع الاسم من الرصّة لكل EndElement، تضمن واجهة برمجة التطبيقات توافق تسلسل StartElement مع الرموز المميزة EndElement حتى في المستندات السيئة التنسيق، علمًا أن التعليقات في المستند يتم تجاهلها. عندما يتلاقى xmlselect مع CharData فإنه يطبع النص فقط في حال احتواء الرصّة على جميع العناصر المسماة من خلال معطيات سطر الأوامر بالترتيب.

الأمر أدناه يطبع نص أي عناصر h2 تظهر أسفل مستويي عناصر div، إن مداخله هي مواصفات XML وهو عبارة عن مستند XML.

```
$ go build gopl.io/ch1/fetch
$ ./fetch http://www.w3.org/TR/2006/REC-xml11-20060816 |
    ./xmlselect div div h2
html body div div h2: 1 Introduction
html body div div h2: 2 Documents
html body div div h2: 3 Logical Structures
html body div div h2: 4 Physical Structures
html body div div h2: 5 Conformance
html body div div h2: 6 Notation
html body div div h2: A References
html body div div h2: B Definitions for Character Normalization
...
```

تمرين 7.17: قم بتطوير xmlselect بحيث تختار العناصر ليس فقط حسب الاسم وإنما حسب الصفات أيضًا، وذلك حسب طريقة صفحات النمط التسلسلية CSS، على سبيل المثال يمكن اختيار عنصر مثل <div id="page" class="wide"> من خلال توافق الرمز التعريفي id أو الصنف class بالإضافة إلى الاسم.

تمرين 7.18: بالاستفادة من مشفر واجهة برمجة التطبيقات المبني على الرمز المميز، قم بكتابة برنامجا يقرأ مستند XML عشوائي ويبني شجرة من العقد العامة لتمثله. العقد nodes نوعان: عقد CharData وتمثل سلاسل النص، وعقدة Element وتمثل العناصر المسماة وسماتهم، كل عقدة عنصر تحتوي على شريحة من العقد التابعة Children node.

قد تستفيد من الإعلانات التالية:

```
import "encoding/xml"

type Node interface{} // CharData or *Element

type CharData string

type Element
 struct {
    Type       xml.Name
    Attr       []xml.Attr
    Children   []Node
}
```

7.15 بعض النصائح

عند تصميم حزمة جديدة، غالبًا يبدأ مبرمجو جو المبتدئون بإنشاء مجموعة من الواجهات ومن ثم تعريف الأنواع المحددة التي ترضي الواجهات، إنن هذا النهج يؤدي إلى الكثير من الواجهات وكل واجهة لديها تطبيق واحد فقط، لا تفعل ذلك! فهذه الواجهات عبارة عن تجريد غير ضروري مما يزيد من فترة التنفيذ. بإمكانك حصر أي طرق النوع أو حقول السلسلة المتاحة خارج الحزمة باستخدام آلية التصدير (راجع القسم 6.6)، إن تحتاج للواجهات فقط عند وجود نوعين محددين أو أكثر لتتعامل معهم بشكل رسمي.

يوجد استثناء لتلك القاعدة وهو عندما يتم إرضاء الواجهة لنوع محدد واحد ولكن لا يمكن للنوع التواجد في نفس الحزمة كالواجهة بسبب تبعياته، في هذه الحالة من الجيد استخدام واجهة لفصل الحزمتين.

وبما أن الواجهات لا تستخدم في جو إلا إذا تم إرضائها بنوعين أو أكثر فمن الضروري إزالة تفاصيل أي تطبيق معين، مما يقلل من حجم الواجهات وتقليل وتبسيط الطرق، وغالبًا تكون طريقة واحدة كما هو الحال في io.Writer أو fmt.Stringer. من السهل إرضاء الواجهات الصغيرة عند تعريف أنواع جديدة، والقاعدة الذهبية لتصميم الواجهات هي أن تلبي الواجهة ما تحتاجه فقط دون إضافات غير ضرورية.

هكذا نكون قد انتهينا من شرح الطرق والواجهات. تدعم جو أسلوب البرمجة الغرضية التوجه ولكن هذا لا يعني أنه يجب عليك استخدام هذا الأسلوب حصريًا، فليس كل شيء يجب أن يكون غرضيا، فالوظائف القائمة بذاتها لها وظيفتها

وكذلك أنواع البيانات المفتوحة. ربما قد لاحظت أن الأمثلة في أول خمسة فصول من الكتاب لا تستدعي أكثر من 20 طريقة مثل input.Sca، على النقيض من الدالة العادية تستدعي fmt.Printf.

8- روتينات جو والقنوات

لم تكن البرمجة المتزامنة أكثر أهمية في أي وقت مضى عنها اليوم، والبرمجة المتزامنة هي تعبير عن البرنامج كتركيب من أنشطة مستقلة متعددة. تتعامل خوادم الويب مع طلبات آلاف العملاء في وقت واحد. وتعرض أجهزة الكمبيوتر اللوحي وتطبيقات الهواتف الذاكية عروض متحركة في واجهة المستخدم بينما تقوم في الوقت ذاته بأداء عمليات حسابية ومعالجة طلبات الشبكة في الخلفية. حتى مشكلات الدُفعة التقليدية - مثل قراءة بعض البيانات، أو الحساب أو كتابة بعض المُخرجات - تستخدم التزامن لإخفاء التأخير في عمليات I/O، واستغلال معالجات الكمبيوتر الحديث المتعددة، والتي يزداد عددها وليس سرعتها كل عام.

تسمح لغة جو بأسلوبين للبرمجة المتزامنة. ويقدم هذا الفصل روتينات جو (goroutines) والقنوات ، التي تدعم عمليات التواصل التتابعية (communicating sequential processes) أو CSP، وهو نموذج تزامن تُمرر فيه القيم بين الأنشطة المستقلة (goroutines)، ولكن تظل المتغيرات مقيدة في أغلب الأحوال بنشاط واحد. يقدم الفصل التاسع بعض جوانب النموذج التقليدي "الخيوط المتعددة في الذاكرة المشتركة" (shared memory multithreading)، والذي سيكون مألوفًا لك لو استخدمت تقنية الخيوط الموجودة في اللغات السائدة الأخرى. سيوضح الفصل التاسع أيضًا بعض المخاطر والعثرات المهمة في البرمجة المتزامنة التي لن نتعرض لها في هذا الفصل.

بالرغم من أن دعم لغة جو للتزامن هو أحد أقوى خصائصها، إلا أن التبرير المنطقي الخاص بالبرامج المتزامنة أصعب بكثير من الخاص بالبرامج التتابعية، كما أن البديهيات التي نكتسبها من البرمجة التتابعية قد تضللنا أحيانًا. لو كانت هذه أول مرة تقابل فيها التزامن، فإننا نوصيك بقضاء القليل من الوقت الإضافي في التفكير في الأمثلة التي سنقدمها في هذين الفصلين.

8.1 روتينات جو Goroutines

يُطلق على كل نشاط مُنفّذ بالتزامن في لغة Go اسم روتين-جو "goroutine". فكر في برنامج يقوم بوظيفتين، واحدة تقوم ببعض الحساب، والأخرى تكتب بعض المُخرجات، وافترض أن كلتا الوظيفتين لا تستدعي الأخرى. قد يستدعي البرنامج التتابعي وظيفة ثم يستدعي الأخرى، ولكن في البرنامج المتزامن الذي يحتوي على اثنين من روتينات جو أو أكثر، يمكن استدعاء كلتا الوظيفتين وتنشيطهما في نفس الوقت. سنرى برنامجا كهذا بعد قليل.

لو كنت استخدمت تقنية خيوط نظام التشغيلي أو تقنيات الخيوط في اللغات أخرى، فإنك ستستطيع الافتراض أن روتين-جو مشابه للخيط، وستتمكن من كتابة برامج صحيحة. ولكن الفارق بين الخيوط و روتينات جو فارق كمي وليس نوعي، وسنصفه بالتفصيل في القسم 9.8.

عندما يبدأ برنامج ما، يكون روتين-جو الوحيد فيه هو الذي يستدعي الوظيفة الأساسية main، وبالتالي يمكن أن نطلق عليه روتين go الرئيسي (main goroutine). تنشأ روتينات جو الجديدة عن طريق عبارة go. ومن الناحية التركيبية، تعتبر عبارة go هي استدعاء وظيفة أو طريقة عادية تسبقها الكلمة المفتاحية go. وتستدعي عبارة go الوظيفة إلى روتين-جو نشأ حديثًا. وتكتمل عبارة go نفسها بشكل فوري:

```
f()              // call f(); wait for it to return
go f()    // create a new goroutine that calls f(); don't wait
```

يحسب روتين-جو الرئيسي في المثال أدناه العدد رقم 45 في متتابعة فيبوناتشي. سيعمل الروتين لمدة طويلة لأنه يستخدم خوارزمية تكرارية عديمة الكفاءة، وبالتالي سنرغب في تقديم مؤشر بصري للمستخدم يوضح أن البرنامج لا يزال يعمل، وهذا من خلال عرض نص متحرك "دوّار":

```
gopl.io/ch8/spinner

func main() {
    go spinner(100 * time.Millisecond)
    const n = 45
    fibN := fib(n) // slow
    fmt.Printf("\rFibonacci(%d) = %d\n", n, fibN)
}

func spinner(delay time.Duration) {
    for {
        for _, r := range `-\|/` {
            fmt.Printf("\r%c", r)
            time.Sleep(delay)
        }
    }
}

func fib(x int) int {
    if x < 2 {
        return x
    }
    return fib(x-1) + fib(x-2)
}
```

بعد ثوان متعددة من عرض النص المتحرك، يعود استدعاء fib(45)، وتطبع الوظيفة الأساسية نتيجتها:

```
Fibonacci(45) = 1134903170
```

تعود الوظيفة الرئيسة بالنتيجة حينها، وعندما يحدث هذا، يتم إنهاء كل روتينات جو فورًا، وينتهي البرنامج. لا توجد طريقة برمجية تمكّن روتين-جو واحد من إيقاف روتين-جو آخر إلا بالعودة من الروتين الرئيسي أو الخروج من البرنامج، ولكن كما سنرى لاحقًا، هناك طريقة للتواصل مع روتين-جو تُمكّنك من أن تطلب منه إيقاف نفسه.

لاحظ كيف يتم التعبير عن البرنامج كتركيب من نشاطين مستقلين، هما الدوران وحساب فيبوناتشي. كل منهما مكتوب كوظيفة مستقلة، ولكن كلاهما يحقق تقدما متزامنا.

8.2 مثال: خادم ساعة متزامن

إن ربط الشبكات هو النطاق الطبيعي المناسب لاستخدام التزامن حيث أن الخوادم عادة ما تتعامل مع صِلات متعددة من عملائها في وقت واحد، حيث يُعتبر كل عميل مستقل بشكل جوهري عن الآخر. سنقدم في هذا القسم حزمة net، التي تقدم مكونات لبناء برامج عميل وخادم مرتبطة شبكيًا، ويتم توصيلها عبر منافذ نطاق TCP أو UDP أو مقبس يونكس. إن حزمة net/http التي استخدمناها منذ الفصل الأول مبنية على وظائف من حزمة net.

إن أول أمثلتنا هو خادم ساعة تتابعي يكتب الوقت الحالي للعميل مرة واحدة كل ثانية:

```
gopl.io/ch8/clock1

// Clock1 is a TCP server that periodically writes the time.
package main
import (
    "io"
    "log"
    "net"
    "time"
)

func main() {
    listener, err := net.Listen("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    for {
        conn, err := listener.Accept()
        if err != nil {
            log.Print(err) // e.g., connection aborted
            continue
        }
        handleConn(conn) // handle one connection at a time
    }
}

func handleConn(c net.Conn) {
    defer c.Close()
```

```
    for {
        _, err := io.WriteString(c, time.Now().Format("15:04:05\n"))
        if err != nil {
            return // e.g., client disconnected
        }
        time.Sleep(1 * time.Second)
    }
}
```

تنشئ وظيفة Listen الكائن net.Listener وهو كائن يستمع للاتصالات الواردة في منفذ الشبكة، وهو في هذه الحالة منذ localhost:8000، TCP. تقوم طريقة Accept الخاصة بالمستمع بإيقاف البرنامج حتى يصلها طلب اتصال، ثم تعيد كائن net.Conn يمثل الاتصال.

تتعامل وظيفة handleConn مع اتصال عميل وحيد. وتعمل في حلقة وتكتب الوقت الحالي، ()time.Now، للعميل. وحيث أن net.Conn ترضي واجهة io.Writer، بالتالي يمكننا الكتابة عليها مباشرة. تنتهي الحلقة عندما تفشل الكتابة، وغالبًا يحدث هذا لأن العميل قطع الاتصال، وعند هذه النقطة، تقوم الوظيفة handleConn بإغلاق الاتصال باستخدام استدعاء "إغلاق/Close" مؤجّل، وتعود لكتابة طلب اتصال آخر.

تقدم طريقة time.Time.Format طريقة لعرض شكل معلومات الوقت والتاريخ وفقًا لمثال. ومعطياتها هي قالب يوضح كيفية صياغة وقت مرجعي، بشكل خاص: Mon Jan 2 03:04:05PM 2006 UTC-0700

إن الوقت المرجعي يحتوي على ثمانية مكونات (يوم الأسبوع، الشهر، تاريخ اليوم في الشهر، إلخ). يمكن أن تظهر أي مجموعة منهم في سلسلة Format بأي ترتيب وبأي عدد من الأشكال، وستعرض المكونات المختارة الخاصة بالوقت والتاريخ في الأشكال المختارة. نحن نستخدم هنا الساعة والدقيقة والثانية في عرض الوقت. وتُعرّف حزمة الوقت قوالب للعديد من أشكال الوقت القياسية، مثل time.RFC1123. تُستخدم نفس الآلية بالعكس عند تحليل الوقت باستخدام time.Parse.

للاتصال بالخادم، ستحتاج إلى برنامج عميل مثل nc (“netcat”)، وهو برنامج منفعة قياسي للتلاعب باتصالات الشبكة:

```
$ go build gopl.io/ch8/clock1
$ ./clockl &
$ nc localhost 8000
13:58:54
13:58:55
13:58:56
13:58:57
^C
```

يعرض العميل الوقت الذي يرسله له الخادم كل ثانية حتى نقاطع العميل بـ Control-C، والذي يتردد صداه في نظم Unix كـ ^C بواسطة الصَدَفة. لو لم تثبّت nc و netcat على نظامك، يمكنك استخدام telnet أو هذه نسخة جو البسيطة من netcat التي تستخدم net.Dial للاتصال بخادم TCP:

```
gopl.io/ch8/netcat1

// Netcat1 is a read-only TCP client.
package main
import (
    "io"
    "log"
    "net"
    "os"
)

func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    mustCopy(os.Stdout, conn)
}

func mustCopy(dst io.Writer, src io.Reader) {
    if _, err := io.Copy(dst, src); err != nil {
        log.Fatal(err)
    }
}
```

يقرأ هذا البرنامج البيانات من الاتصال ويكتبه على المُخرَج القياسي حتى تحدث حالة نهاية-الملف أو يقع خطأ. إن وظيفة mustCopy هي وسيلة تُستخدم في العديد من الأمثلة في هذا القسم. لنشغّل عميلين في نفس الوقت في محطات مختلفة، واحدة تظهر على اليسار والأخرى تظهر على اليمين:

```
$ go build gopl.io/ch8/netcat1
$ ./netcat1
13:58:54                                 $ ./netcat1
13:58:55
13:58:56
^C
                                    13:58:57
                                    13:58:58
                                    13:58:59
                                    ^C
$ killall clock1
```

إن الأمر killall هو وسيلة في Unix تقتل كل العمليات ذات الاسم المحدد.

يجب أن ينتظر العميل الثاني حتى انتهاء العميل الأول لأن الخادم تتابعي، وهو يتعامل مع عميل واحد فقط في المرة الواحدة. هناك تغيير واحد فقط مطلوب لجعل الخادم متزامن: إضافة كلمة go المفتاحية لاستدعاء handleConn يجعل كل استدعاء يعمل في روتين-جو خاص به.

```
gopl.io/ch8/clock2 for

for {
    conn, err := listener.Accept()
    if err != nil {
        log.Print(err) // e.g., connection aborted
        continue
    }
    go handleConn(conn) // handle connections concurrently
}
```

الآن، يمكن أن يتلقى عملاء متعددون الوقت مرة واحدة:

```
$ go build gopl.io/ch8/clock2
$ ./clock2 &
$ go build gopl.io/ch8/netcat1
$ ./netcat1
14:02:54                $ ./netcat1
14:02:55                14:02:55
14:02:56                14:02:56
14:02:57                ^C
14:02:58
14:02:59                $ ./netcat1
14:03:00                14:03:00
14:03:01                14:03:01
^C                      14:03:02
                        ^C
$ killall clock2
```

تمرين 8.1: عدّل clock2 ليقبل رقم المنفذ، واكتب برنامجا، clockwall، يعمل كعميل لعدة خوادم ساعة في وقت واحد، ويقرأ الأوقات من كل واحد منها ويعرض النتائج في جدول، يشبه حائط الساعات الذي نراه في بعض مكاتب الأعمال. لو كان بإمكانك الوصول لأجهزة كمبيوتر موزعة بشكل جغرافي، فقم بإجراء هذه الأمثلة عن بعد، أو قم بإجراء الأمثلة محليًا في منافذ مختلفة ذات مناطق زمنية زائفة.

```
$   TZ=US/Eastern    ./clock2   -port 8010  &
$   TZ=Asia/Tokyo    ./clock2   -port 8020  &
$   TZ=Europe/London ./clock2   -port 8030  &
$ clockwall NewYork=localhost:8010 London=localhost:8020 Tokyo=localhost:8030
```

تمرين 8.2: قم بكتابة خادم بروتوكول نقل الملف (FTP) المتزامن. يجب أن يفسر الخادم الأوامر من كل عميل مثل cd لتغيير المسار، و ls لإدراج دليل، و get لإرسال محتويات ملف، و close لإغلاق الاتصال. يمكنك استخدام أمر ftp القياسي كعميل، أو كتابة أمرك الخاص.

8.3 مثال: خادم صدى متزامن

لقد استخدم خادم الساعة روتين-جو واحدا لكل اتصال. وسنبني في هذا القسم خادم صدى يستخدم روتينات جو متعددة في كل اتصال. تكتب معظم خوادم الصدى ما تقرأه وحسب، وهو ما يمكن تحقيقه بهذه النسخة التافهة من handleConn:

```
func handleConn(c net.Conn) {
     io.Copy(c, c) // NOTE: ignoring errors c.Close()
}
```

يمكن لخادم الصدى الأكثر إثارة للاهتمام أن يحاكي أصداء الصدى الحقيقي، باستجابة صاخبة أولاً ("HELLO!")، ثم معتدلة ("Hello!") بعد تأخير قليل، ثم ("hello!") هادئة قبل أن يتلاشى إلى لا شيء، كما هو الحال في هذه النسخة من handleConn:

```
gopl.io/ch8/reverb1

func echo(c net.Conn, shout string, delay time.Duration) {
    fmt.Fprintln(c, "\t", strings.ToUpper(shout))
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", shout)
    time.Sleep(delay)
    fmt.Fprintln(c, "\t", strings.ToLower(shout))
}

func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        echo(c, input.Text(), 1*time.Second)
    }
    // NOTE: ignoring potential errors from input.Err()
    c.Close()
}
```

سنحتاج إلى تحديث برنامج عميلنا حتى يرسل مُدخل سطر الأوامر إلى الخادم أثناء نسخ استجابة الخادم إلى المُخرَج، وهو ما يقدم فرصة أخرى لاستخدام التزامن:

```
gopl.io/ch8/netcat2

func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    go mustCopy(os.Stdout, conn)
    mustCopy(conn, os.Stdin)
}
```

بينما يقرأ روتين-جو الرئيسي المُدخل القياسي ويرسله إلى الخادم، هناك روتين-جو ثاني يقرأ ويطبع استجابة الخادم. عندما يواجه روتين-جو الرئيسي نهاية المُدخل، مثلا بعد أن يكتب المستخدم Control-D(^D) في سطر الأوامر (أو Control-Z المكافئة لها في مايكروسوفت ويندوز)، يتوقف البرنامج، حتى لو كان روتين-جو الآخر لا يزال لديه عمل يجب أن يقوم به. (سنرى كيف سنجعل البرنامج ينتظر انتهاء كلا الجانبين قبل التوقف عندما نتحدث عن القنوات في القسم 8.4.1).

إن مُدخل العميل في الجلسة أدناه ذات محاذاة لليسار، بينما استجابات الخادم مُزاحة. ويصيح العميل في خادم الصدى ثلاث مرات:

```
$ go build gopl.io/ch8/reverb1
$ ./reverbl &
$ go build gopl.io/ch8/netcat2
$ ./netcat2
Hello?
    HELLO?
    Hello?
    hello?
Is there anybody there?
    IS THERE ANYBODY THERE?
Yooo-hooo!
    Is there anybody there?
    is there anybody there?
    YOOO-HOOO!
    Yooo-hooo!
    yooo-hooo!
^D
$ killall reverbl
```

لاحظ أن الصيحة الثالثة من العميل لم يتعامل معها إلا بعد تلاشي الصيحة لثانية، وهو أمر غير واقعي. سيتكوّن الصدى الحقيقي من "تركيب/Composition" من ثلاث صيحات مستقلة. ولمحاكاتها، سنحتاج للمزيد من روتينات جو. ومرة أخرى، كل ما سنحتاج لفعله هو إضافة كلمة go المفتاحية، وهذه المرة سنضيفها لاستدعاء الصدى echo:

```
gopl.io/ch8/reverb2

func handleConn(c net.Conn) {
    input := bufio.NewScanner(c)
    for input.Scan() {
        go echo(c, input.Text(), 1*time.Second)
    }
    // NOTE: ignoring potential errors from input.Err()
    c.Close()
}
```

إن معطيات الوظيفة التي تبدأ بـ go تُقيّم عند تنفيذ عبارة go نفسها، بالتالي تُقيّم input.Text() في روتين-جو الرئيسي.

أصبحت الأصداء متزامنة الآن ومتداخلة زمنيًا:

```
$ go build gopl.io/ch8/reverb2
$ ./reverb2 &
$ ./netcat2
Is there anybody there?
    IS THERE ANYBODY THERE?
Yooo-hooo!
    Is there anybody there?
    YOOO-HOOO!
    is there anybody there?
    Yooo-hooo!
    yooo-hooo!
^D
$ killall reverb2
```

كان كل هذا مطلوب لجعل الخادم يستخدم التزامن، وكان إدخال كلمتيّ go مفتاحيتين مهم ليس فقط للتعامل مع الاتصالات من عملاء متعددين، ولكن أيضًا للعمل داخل الاتصال الواحد.

مع ذلك، عند إضافة هذه الكلمات المفتاحية، علينا أن ندرس بعناية أنه من الآن يصبح استدعاء طُرق net.Conn بالتزامن، وهو أمر غير صحيح في معظم الأنواع. سنناقش مفهوم "أمان التزامن/ concurrency safety" هذا في الفصل التالي.

8.4 القنوات/Channels

لو كانت روتينات جو هي أنشطة برنامج Go المتزامن، فإن القنوات هي الاتصالات بين هذه الأنشطة. إن القناة هي آلية اتصال تسمح لروتين-جو واحد يرسل قيم إلى روتين-جو آخر. وكل قناة هي مجرى لقيم نوع محدد، يُطلق عليه نوع العنصر "element type" الخاص بالقناة. إن نوع القناة التي تحتوي عناصرها على النوع int تُكتب chan int.

نستخدم وظيفة make مدمجة لصنع القناة:

```
ch := make(chan int) // ch has type 'chan int'
```

كما هو الحال مع الخرائط، فإن القناة هي مرجع (reference) لبنية البيانات مصنوعة باستخدام make. عندما ننسخ قناة أو نمرر واحدة كمعطى لوظيفة، فإننا نسخ مرجعا، حتى يرجع المُستدْعِي والمُستدْعَى إلى نفس بنية البيانات. وكما هو الحال مع أنواع المراجع الأخرى، ستكون القيمة الصفرية للقناة هي nil.

يمكن مقارنة قناتين من نفس النوع باستخدام ==. وتكون المقارنة صحيحة لو كان كلاهما مراجع لنفس بنية بيانات القناة. يمكن مقارنة القناة مع nil أيضًا.

تمتلك القناة عمليتين رئيسيين، هما "إرسال" (Send) و"استقبال" (receive)، ويُطلق عليهما إجمالا "الاتصالات" (communications). تنقل عبارة الإرسال قيمة من روتين-جو إلى آخر عبر القناة، وتنفذ تعبير الاستقبال المُناظِر. تُكتب كلتا العمليتين باستخدام عامل ->، وفي عبارة الإرسال، يفصل -> القناة عن معامِلات القيمة. أما في تعبير الاستقبال، فإن -> يسبق معامِل القناة. إن تعبير الاستقبال الذي لا تُستخدم نتيجته هو عبارة صحيحة.

```
ch <- x // عبارة إرسال
x = <-ch // تعبير استقبال في عبارة إسناد
<-ch // عبارة استقبال، يتم التخلص من النتيجة
```

تدعم القناة عملية ثالثة هي "إغلاق" (Close)، والتي ترسل تنبيه يشير إلى أنه لن ترسل أي قيم عبر هذه القناة أبدًا، وأي محاولات تالية للإرسال ستفشل. ينتج عن عمليات الاستقبال في قناة مغلقة قيم أُرسلت حتى لم يعد هناك قيم باقية، وأي عمليات استقبال بعدها ستكتمل مباشرة، وينتج عنها قيمة صفرية في نوع عنصر القناة.

نستدعي وظيفة close المدمجة لإغلاق القناة:

```
close(ch)
```

إن القناة المصنوعة باستدعاء بسيط لـ make يُطلق عليها قناة غير صِوانيّة (unbuffered channel)، ولكن make يقبل معطى ثاني اختياري، رقم صحيح يُطلق عليه "سعة" القناة (capacity). لو كانت السعة غير صفرية، فإن make يصنع قناة صِوانية (buffered channel).

```
ch = make(chan int) // unbuffered channel
ch = make(chan int, 0) // unbuffered channel
ch = make(chan int, 3) // buffered channel with capacity 3
```

سنلقي نظرة على القنوات غير الصِوانية أولا، ثم سنتحدث عن القنوات الصوانية في القسم 8.4.4.

8.4.1 القنوات غير الصِوانية

إن عملية الإرسال في قناة غير صِوانية تحجب إرسال روتين-جو حتى ينفذ روتين-جو آخر استقبال مناظِر في نفس القناة، وعند هذه النقطة، تُنقل القيمة، ويمكن لكلا روتينان جو المواصلة. على العكس، لو أُجريت عملية الاستقبال أولا، فإن روتين-جو المستقبِل سيُحجَب حتى يؤدي روتين-جو الآخر الإرسال على نفس القناة.

يجعل الاتصال عبر قناة غير صوانية روتينات جو الإرسال والاستقبال تتزامن (synchronize). وبسبب هذا، يُطلق على القنوات غير الصِوانية أحيانًا اسم قنوات تزامنية (synchronous channels). عندما تُرسل قيمة إلى قناة غير صِوانية، فإن استقبال القيمة يحدث قبل إعادة تفعيل روتين-جو المُرسِل.

عندما نقول أن x يحدث قبل y في نقاشات التزامن، لا نعني وحسب أن x يحدث في فترة زمنية سابقة لـ y، بل نعني أننا نضمن أنه سيفعل هذا، وأن كل آثاره السابقة، مثل تحديثات المتغيرات، مكتملة، ويمكنك الاعتماد عليها.

عندما لا تحدث x قبل ولا بعد y، نقول أن x متزامنة مع y. لا يعني هذا أن x و y متزامنين بالضرورة، بل معناه فقط أننا لا يمكننا افتراض أي شيء عن ترتيب حدوثهما. وكما سنرى في الفصل التالي، من الضروري ترتيب أحداث معينة خلال تنفيذ البرنامج لتجنب المشكلات الناتجة عن وصول اثنين من روتينات جو إلى نفس المتغير في وقت متزامن.

ينسخ برنامج العميل الموجود في القسم 8.3 المُدخل إلى الخادم في روتين-جو الرئيسي الخاص به، وبالتالي ينتهي برنامج العميل بمجرد إغلاق تيار المُدخل، حتى لو كان هناك روتين-جو لا يزال يعمل في الخلفية. ولجعل البرنامج ينتظر اكتمال روتين-جو الذي في الخلفية قبل الخروج من البرنامج، نستخدم قناة لمزامنة كلا روتينيّ جو:

```go
gopl.io/ch8/netcat3

func main() {
    conn, err := net.Dial("tcp", "localhost:8000")
    if err != nil {
        log.Fatal(err)
    }
    done := make(chan struct{})
    go func() {
        io.Copy(os.Stdout, conn) // NOTE: ignoring errors
        log.Println("done")
        done <- struct{}{} // signal the main goroutine
    }()
    mustCopy(conn, os.Stdin)
    conn.Close()
    <-done // wait for background goroutine to finish
}
```

عندما يُغلق المستخدم تيار المُدخل القياسي، يعود mustCopy، ويستدعي روتين-جو الرئيسي conn.Close()، ويغلق كلا نصفيّ اتصال الشبكة. إن إغلاق نصف الكتابة في الاتصال يجعل الخادم يرى حالة "نهاية-الملف"، أما إغلاق نصف

القراءة فيجعل روتين-جو الخلفية يستدعي io.Copy ليسجل خطأ "القراءة من اتصال مغلق"، ولهذا السبب حذفنا سجل الخطأ. يقترح التمرين 8.3 حلا أفضل. (لاحظ أن عبارة go تستدعي وظيفة حرفية، بنية مشتركة).

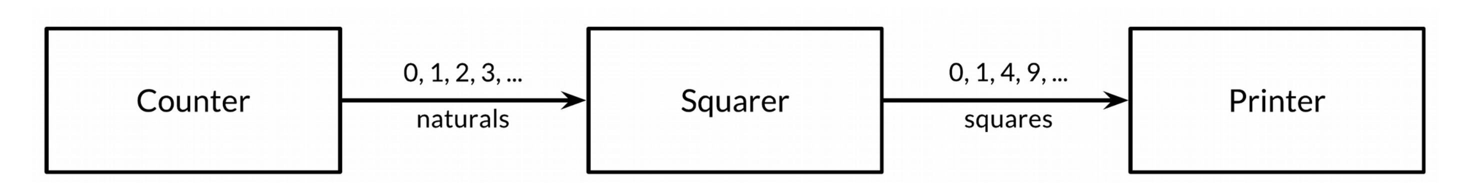
يسجل روتين-جو الخلفية رسالة قبل أن يعود، ثم يرسل قيمة إلى القناة المنتهية. ينتظر روتين-جو الرئيسي حتى يستلم تلك القيمة قبل العودة. ونتيجة لهذا، يسجل البرنامج دائمًا رسالة "تم" قبل الخروج.

إن الرسائل التي تُرسل عبر القنوات لها جانبين مهمين. تمتلك كل رسالة قيمة، ولكن أحيانًا ما تكون حقيقة الاتصال واللحظة التي يحدث فيها مهمين بنفس القدر. نحن نطلق على الرسائل "أحداث" (events) عندما نرغب في التأكيد على هذا الجانب. عندما لا يحمل الحدث معلومات إضافية، أي عندما يكون هدفه الوحيد هو التزامن، فسنؤكد على هذا باستخدام القناة التي نوع عنصرها هو {}struct، بالرغم من شيوع استخدام قناة bool أو int لتحقيق نفس الغرض، حيث أن done <- 1 أقصر من done <- struct{}{} .

تمرين 8.3: في netcat3، تمتلك قيمة الواجهة conn النوع المحدد net.TCPConn*، والذي يمثل اتصال TCP. يتكوّن اتصال TCP من نصفين يمكن إغلاقهما بشكل مستقل عن بعضهما باستخدام طرقتيِ CloseRead و CloseWrite. عدّل روتين-جو الرئيسي في netcat3 لإغلاق نصف الكتابة فقط في الاتصال حتى يستمر البرنامج في طباعة الأصداء الأخيرة من خادم reverbl حتى بعد إغلاق المُدخل القياسي. (إن فعل هذا في خادم reverb2 أصعب، انظر التمرين 8.4).

8.4.2 التواردات (Pipelines)

يمكن استخدام القنوات لربط روتينات جو معًا بحيث يصبح مُخرَج أحدهما مُدخل لآخر. يُطلق على هذا "التوارد" (pipeline). يتكون البرنامج أدناه من ثلاث روتينات جو مرتبطة معًا عبر قناتين كما هو موضح تخطيطيًا في الشكل 8.1.



الشكل 8.1: توارُد من ثلاث مراحل.

إن أول روتين-جو، وهو "counter" ينتج الأعداد الصحية 0، 1، 2، ...، ويرسلها إلى قناة إلى روتين-جو الثاني، وهو "squarer"، والذي يستقبل كل قيمة، ويربّعها، ويرسل النتيجة إلى قناة أخرى إلى روتين-جو ثالث هو "printer"، والذي يستلم القيم التربيعية ويطبعها. ولتوضيح هذا المثال، اخترنا عمدًا وظائف بسيطة جدًا، بالرغم من أنها عمليات حسابيةلا تستحق روتينات جو خاصة بها في أي برنامج واقعي.

```go
gopl.io/ch8/pipeline1

func main() {
    naturals := make(chan int)
    squares := make(chan int)
    // Counter
    go func() {
        for x := 0; ; x++ {
            naturals <- x
        }
    }()
    // Squarer
    go func() {
        for {
            x := <-naturals
            squares <- x * x
        }
    }()
    // Printer (in main goroutine)
    for {
        fmt.Println(<-squares)
    }
}
```

كما يمكن أن تتوقع، يطبع البرنامج سلسلة لا نهائية من المربعات 0، 1، 4، 9، إلخ. وتوارد كهذا يمكن إيجاده في برامج الخادم التي تعمل لفترات طويلة، والتي تُستخدم القنوات فيها للتواصل مدى الحياة بين روتينات جو تحتوي على حلقات لا نهائية. ولكن ماذا لو أردنا إرسال عدد محدد من القيم عبر التوارد فقط؟

لو كان المُرسِل يعلم أنه لن تُرسل قيم أخرى أبدًا في قناة، فمن المفيد توصيل هذه الحقيقة إلى روتينات جو المُستقبِلة حتى يمكنها إيقاف الانتظار. يتحقق هذا من خلال إغلاق (closing) القناة باستخدام وظيفة close المدمجة:

```go
close(naturals)
```

بعد إغلاق لقناة، ستحدث حالة هلع في أي عمليات إرسال أخرى عليها. بعد استنزاف القناة المغلقة، أي بعد استقبال آخر عنصر مُرسَل، ستتواصل كل عمليات الاستقبال التالية دون حجب ولكنها ستنتج قيمة صفرية. إن إغلاق قناة naturals أعلاه سيجعل حلقة squarer تدور حيث أنها تستقبل تيار لا ينتهي من القيم الصفرية، وسترسل هذه الأصفار إلى الطابعة.

لا توجد طريقة لإجراء اختبار مباشر لمعرفة إذا كانت القناة قد أُغلقت أم لا، ولكن هناك تنويع في عملية الاستقبال يخرج نتيجتين: عنصر القناة المستقبِلة، بالإضافة إلى القيمة المنطقية (boolean value)، والتي يُطلق عليها "ok"، وهذا صحيح بالنسبة للاستقبال الناجح، وخاطئ بالنسبة للاستقبال في قناة مغلقة ومستنزَفة. باستخدام هذه الخاصية، يمكننا تعديل حلقة squarer لتتوقف عندما تُستنزَف قناة naturals، وتغلق قناة squares بدورها.

```go
// Squarer
```

```
go func() {
     for {
          x, ok := <-naturals
          if !ok {
               break // channel was closed and drained
          }
          squares <- x * x
     }
     close(squares)
}()
```

نظرًا لكون البِنية "syntax" الموضحة أعلاه فوضوية، وهذا النمط شائع، تتركنا اللغة نستخدم حلقة range للمرور على كل القنوات أيضًا. هذه بنية ملائمة أكثر لاستقبال كل القيم المُرسَلة على قناة، ثم إنهاء الحلقة بعد الانتهاء من آخر قيمة.

في التوارد أدناه، عندما ينتهي روتين-جو من نوع counter بإنهاء حلقته بعد 100 عنصر، فإنه يغلق قناة naturals، ويجعل squarer تنهي حلقتها وتغلق وظيفة squarer. (في البرامج الأكثر تعقيدًا، قد يكون من المنطقي أن تقوم وظائف counter و squarer بتأجيل طلبات الإغلاق حتى النهاية). أخيرًا، ينهي روتين-جو الرئيس حلقته ويخرج من البرنامج.

```
gopl.io/ch8/pipeline2

func main() {
    naturals := make(chan int)
    squares := make(chan int)
    // Counter
    go func() {
        for x := 0; x < 100; x++ {
            naturals <- x
        }
        close(naturals)
    }()
    // Squarer
    go func() {
        for x := range naturals {
            squares <- x * x
        }
        close(squares)
    }()
    // Printer (in main goroutine)
    for x := range squares {
        fmt.Println(x)
    }
}
```

أنت لا تحتاج إلى إغلاق كل قناة انتهيت منها. من الضروري فقط إغلاق القناة عندما يكن من المهم إخبار روتينات جو المستقبِلة أن كل البيانات أرسِلت. إن القناة التي يقرر جامع القمامة أنها لا يمكن الوصول إليها سيتم الاستحواذ على كل مواردها سواء أُغلقت أم لا. (لا تخلط بين هذا وبين عملية إغلاق الملفات المفتوحة. من المهم استدعاء طريقة Close في كل ملف عند الانتهاء منه).

إن محاولة إغلاق قناة مغلقة بالفعل يُحدِث هلعا، وكذلك محاولة إغلاق قناة nil. إن إغلاق القنوات له فائدة أخرى كآلية بث، وسنتحدث عن هذا الأمر في القسم 8.9.

8.4.3 أنواع القناة أحادية الاتجاه/Unidirectional Channel Types

يُعتبر من الطبيعي تقسيم الوظائف الكبيرة إلى قطع أصغر مع نمو البرنامج. وقد استخدم برنامجنا السابقة ثلاثة روتينات جو، وتواصل عبر قناتين، واللتان كانتا متغيرات محلية في الوظيفة الرئيسية. وقد انقسم البرنامج بطبيعة الحال إلى ثلاث وظائف:

```
func counter(out chan int)
func squarer(out, in chan int)
func printer(in chan int)
```

إن وظيفة squarer – الموجود في وسط التوارد – تأخذ مُعامِلين هما قناة المُدخَل وقناة المُخرج. يمتلك كلاهما نفس النوع، ولكن استخداماتهما متضادة: فقناة in يتم الاستقبال منها فقط، وقناة out يتم الإرسال إليها فقط. توصل الأسماء in و out هذه الفكرة، ولكن مع ذلك، لا يوجد شيء يمنع squarer من الإرسال إلى in أو الاستقبال من out.

إن هذا الترتيب تقليدي. عندما تزود قناة كمعامل وظيفة، فسيتم هذا بطريقة دائمة بهدف استخدامها حصريًا للإرسال أو حصريًا للاستقبال.

لتوثيق هذا الهدف ومنع سوء الاستخدام، يقدم نظام أنواع جو أنواع قناة أحادية الاتجاه تعرض جهة واحدة فقط أو الجهة الأخرى من عمليات الإرسال أو الاستقبال. إن النوع chan<- int وهو قناة إرسال فقط لـ int – يسمح بالإرسال ولكن لا يسمح بالاستقبال. وعلى النقيض، النوع <-chan int ، هو قناة استقبال فقط وليس إرسال. (إن مكان السهم -> بالنسبة لكلمة chan المفتاحية هو رمز تذكُري). تكتشف انتهاكات هذا النظام في وقت التجميع/الترجمة.

نظرًا لكون عملية الإغلاق close تؤكد على أنه لن يحدث أي إرسال آخر في القناة، فإن روتين-جو المرسِل فقط هو الذي سيظل في موضع يمكن استدعاءه منه، ولهذا لسبب تُعَد محاولة إغلاق قناة استقبال فقط هي أحد أخطاء وقت الترجمة.

إليك توارد التربيع مرة أخرى، ولكن هذه المرة مع أنواع قناة أحادية الاتجاه:

```
gopl.io/ch8/pipeline3

func counter(out chan<- int) {
    for x := 0; x < 100; x++ {
        out <- x
    }
    close(out)
```

```go
}

func squarer(out chan<- int, in <-chan int) {
    for v := range in {
        out <- v * v
    }
    close(out)
}

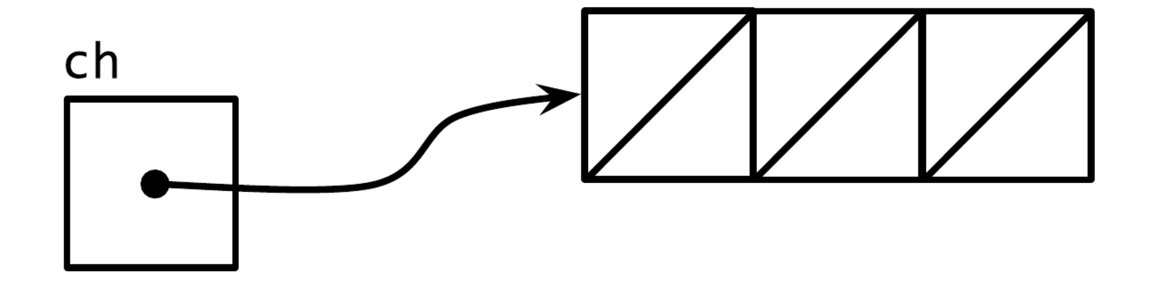
func printer(in <-chan int) {
    for v := range in {
        fmt.Println(v)
    }
}
func main() {
    naturals := make(chan int)
    squares := make(chan int)
    go counter(naturals)
    go squarer(squares, naturals)
    printer(squares)
}
```

إن الاستدعاء counter(naturals) يحول naturals بشكل ضمني، حيث يحول قيمة من النوع chan int إلى نوع المعامل chan<- int. ويقوم استدعاء printer(squares) بتحويل ضمني مشابه إلى chan int->. إن التحويلات من أنواع القناة ثنائية الاتجاه إلى أحادية الاتجاه مسموح به في أي مهمة. مع ذلك، لا يوجد رجوع، بمجرد أن يكون لديك قيمة نوع أحادي الاتجاه مثل chan<- int، لن تكون هناك طريقة للحصول على قيمة من نوع chan int منه تشير إلى نفس بنية بيانات القناة.

8.4.4 القنوات الصِوانية / Buffered Channels

تمتلك القناة الصِوانية طابورا من العناصر. ويتحدد أقصى حجم للطابور عند صنعه من خلال مُعطى السعة في make. تصنع العبارة أدناه قناة صِوانية قادرة على حمل ثلاث قيم متسلسلة. ويقدم الشكل 8.2 تمثيلا رسوميا لـ ch والقناة التي يشير إليها.

```go
ch = make(chan string, 3)
```



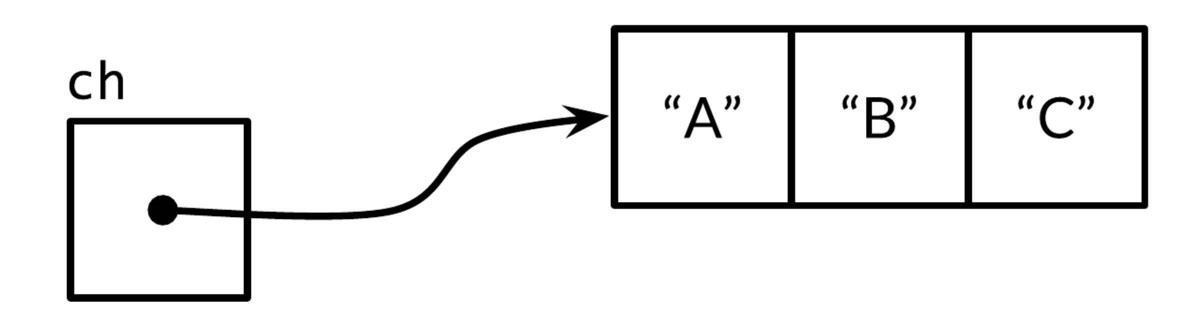
الشكل 8.2: قناة صِوانية فارغة.

تُدخل عملية الإرسال في القناة الصِوانية عنصرا في مؤخرة الطابور، بينما تحذف عملية الاستقبال عنصر من مقدمته. لو كانت القناة ممتلئة، فإن قناة الإرسال تحجب روتين-جو الخاصة بها حتى يصبح هناك مكانا متاحا من خلال استقبال روتين-جو آخر لقيمة أخرى. على النقيض، لو كانت القناة فارغة، فإن عملية الاستقبال تُحجب حتى تُرسل قيمة بواسطة روتين-جو آخر.

يمكننا إرسال ما يصل إلى ثلاث قيم إلى هذه القناة بدون حجب روتين-جو:

```
ch <- "A"
ch <- "B"
ch <- "C"
```

بعدها تصبح القناة ممتلئة (انظر الشكل 8.3)، وستُحجب عبارة الإرسال الرابعة.

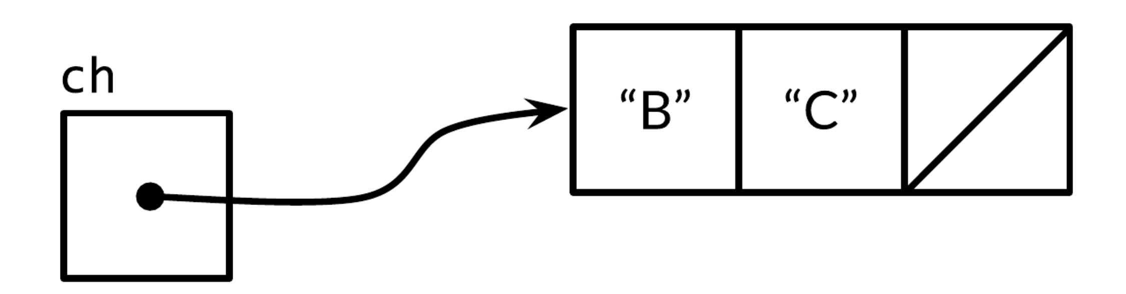


الشكل 8.3: قناة صِوانية ممتلئة.

لو تلقينا قيمة واحدة،

```
fmt.Println(<-ch) // "A"
```

فلن تكون القناة ممتلئة ولا فارغة (الشكل 8.4)، لذا يمكن أن تستمر عملية الإرسال والاستقبال دون حجب. بهذه الطريقة، يفصل صوان القناة بين روتينات جو الإرسال والاستقبال.



الشكل 8.4: قناة صِوانية ممتلئة جزئيًا.

في الحالة غير المرجحة التي يحتاج فيها البرنامج لمعرفة سعة صوان القناة، يمكن الحصول عليه من خلال استدعاء وظيفة cap مدمجة:

```
fmt.Println(cap(ch)) // "3"
```

عند تطبيقها على القناة، تعيد وظيفة len المدمجة عدد العناصر المُصانة حاليًا. وحيث أن هذه المعلومات من المرجح أن تصبح قديمة بمجرد استقبالها في البرنامج المتزامن، تصبح قيمتها محدودة، ولكنها يمكن أن تكون مفيدة خلال تشخيص الخطأ أو تحسين الأداء.

```
fmt.Println(len(ch)) // "2"
```

بعد عمليتيّ استقبال أخرتين، تصبح القناة فارغة مرة أخرى، وستحجب الرابعة:

```
fmt.Println(<-ch) // "B"
fmt.Println(<-ch) // "C"
```

أجريت عمليات الإرسال والاستقبال في هذا المثال بواسطة نفس روتين-جو، ولكن في البرامج الحقيقية تُنفذ بواسطة روتينات جو مختلفة عادة. يشعر المبتدئون بالإغراء عادة لاستخدام القنوات الصِوانية داخل روتين-جو واحد كصف، ويغريهم البنية البسيطة السهلة، ولكن هذا خطأ. إن القنوات مرتبطة بعمق بالجدولة الزمنية لروتينات جو، وبدون روتين-جو آخر يستقبل من القناة، فإن الراسل – وربما البرنامج بأكمله – يخاطر بأن يتعرض للحجب للأبد. لو كان كل ما تحتاجه هو صف بسيط، فاصنع واحدا باستخدام شريحة.

يوضح المثال أدناه تطبيقا للقناة الصوانية. وهو يقدم طلبات موازية لثلاث مرايا (mirrors)، أي خوادم متكافئة ولكنها موزعة جغرافيًا. وهو يرسل إجاباتها عبر قناة صوانية، ثم يستقبل ويعيد الاستجابة الأولى فقط، وهي أسرع استجابة تصل. من ثم، فإن mirroredQuery يعيد النتيجة قبل أن يستجيب الخادمين الأبطأ حتى. (من الطبيعي إلى حد كبير أن ترسل العديد من روتينات جو قيما إلى نفس القناة بالتزامن، كما هو موضح في هذا المثال، أو أن تستقبل من نفس القناة).

```
func mirroredQuery() string {
     responses := make(chan string, 3)
     go func() { responses <- request("asia.gopl.io") }()
     go func() { responses <- request("europe.gopl.io") }()
     go func() { responses <- request("americas.gopl.io") }()
     return <-responses // return the quickest response
}
func request(hostname string) (response string) { /* ... */ }
```

لو كنا استخدمنا قناة غير صِوانية، فإن الروتينين الأبطأ كانا سيصبحان عالقين أثناء إرسال استجاباتهم عبر القناة التي لا يوجد أي روتين-جو آخر ليستقبلها. وهذا الموقف يُطلق عليه "تسريب روتين-جو " (goroutine leak)، وسيمثل خطأ

في البرمجة. وعلى العكس من متغيرات القمامة، لا يتم جمع روتينات جو المسربة تلقائيا، لذا من المهم التأكد أن تنهي روتينات جو نفسها عندما لا يعود هناك حاجة إليها.

إن الاختيار بين القنوات الصِوانية وغير الصِوانية، واختيار سعة القناة الصِوانية يمكن أن يؤثر على صحة البرنامج. تقدم القنوات غير الصوانية ضمانات تزامن أقوى لأن كل عملية إرسال تتزامن مع عملية الاستقبال المقابلة لها، بينما في القنوات الصوانية، تنفصل تلك العمليات عن بعضها البعض. كذلك، عندما نعرف الحد الأعلى لعدد القيم التي سيتم إرسالها في قناة، سيكون من المعتاد صنع قناة صوانية بهذا الحجم وأداء كل عمليات الإرسال قبل استقبال القيمة الأولى. إن الفشل في تخصيص سعة صوان كافية سيُدخِل البرنامج إلى نهاية مغلقة.

يمكن لصوانية القناة التأثير على أداء البرنامج أيضًا. تخيل ثلاث طُهاة في متجر للكعك، أحدهما يخبز، والثالث يزيّن الكعكة، والثالث يقطع كل كعكة قبل تمريرها إلى الطاهي التالي له في خط التجميع. لو كانوا في مطبخ ضيق، فإن كل طاهي انتهى من كعكة يجب أن ينتظر حتى يستعد الطاهي التالي لقبولها، وهذا هو الحال في الاتصال عبر قناة غير صوانية.

لو كانت هناك مساحة تَسَع كعكة واحدة بين كل طاهي وآخر، فإن الطاهي يمكنه وضع الكعكة المنتهية فيها وبدء العمل على الكعكة التالية فورًا، وهذا هو الوضع في القناة الصوانية ذات السعة 1. طالما أن الطهاة يعملون بنفس السرعة في المتوسط، فإن معظم هذه التسليمات ستنتهي سريعًا، وستجعل الاختلافات بين سرعتهم أكثر سلاسة. إن وجود مساحة أكبر بين الطهاة – صوانات أكبر – سيسهل الاختلافات العابرة الأكبر في سرعاتهم بدون تأخير خط التجميع، يحدث هذا عندما يأخذ طاهي فترة راحة قصيرة، ثم يسرع بالعودة للحاق بهم.

من ناحية أخرى، لو كانت مرحلة مبكرة في خط التجميع أسرع باستمرار من المرحلة التالية، فإن الصوان بينهما سيصبح ممتلئ معظم الوقت. على النقيض، لو كانت مرحلة متأخرة أسرع، فإن الصوان سيكون فارغ عادة. إن الصوان لا يقدم أي فائدة في هذه الحالة.

إن تشبيه خط التجميع هو تشبيه مفيد للقنوات و روتينات جو. على سبيل المثال، لو كانت المرحلة الثانية دقيقة وتفصيلية أكثر، فقد لا يتمكن طاهي واحد من مواكبة الإمداد من الطاهي الأول أو تلبية طلب الطاهي الثالث. ولحل هذه المشكلة، يمكننا تعيين طاهي آخر لمساعدة الطاهي الثاني، يقوم بنفس المهمة ولكن يعمل بشكل مستقل. هذا شبيه بصنع روتين-جو آخر يتواصل عبر نفس القنوات.

ليس لدينا مساحة لإظهار هذا هنا، ولكن حزمة gopl.io/ch8/cake تحاكي متجر الكعك هذا، وبها العديد من المعلمات التي يمكنك تغييرها. وهي تشمل قياسات أداء (انظر 11.4) لعدد قليل من السيناريوهات المذكورة أعلاه.

8.5 تكرار الحلقات بالتوازي

سنستكشف في هذا القسم بعض أنماط التزامن الشائعة لتنفيذ كل تكرارات الحلقة بالتوازي. وسندرس مشكلة إنتاج صور مُصغرة من مجموعة من الصور ذات الحجم الكامل. تقدم حزمة gopl.io/ch8/thumbnail وظيفة ImageFile التي يمكنها تحجيم صورة واحدة. لن نقدم طريقة تطبيقها هنا ولكن يمكنك تحميلها من gopl.io.

```
gopl.io/ch8/thumbnail

package thumbnail

// ImageFile reads an image from infile and writes
// a thumbnail-size version of it in the same directory.
// It returns the generated file name, e.g. "foo.thumb.jpeg".
func ImageFile(infile string) (string, error)
```

يمر البرنامج أدناه على قائمة بأسماء الصور وينتج صورة مصغرة لكل منها:

```
gopl.io/ch8/thumbnail

// makeThumbnails makes thumbnails of the specified files.
func makeThumbnails(filenames []string) {
    for _, f := range filenames {
        if _, err := thumbnail.ImageFile(f); err != nil {
            log.Println(err)
        }
    }
}
```

من الواضح أن ترتيب معالجتنا للملفات غير مهم، حيث أن كل عملية تحجيم مستقلة عن العمليات الأخرى. والمشاكل -التي من هذا النوع - التي تتكون بالكامل من مشكلات فرعية مستقلة تمامًا عن بعضها توصف بأنها "متوازية بشكل مُحرِج" (embarrassingly parallel). إن المشكلات المتوازية بشكل مُحرِج هي أسهل نوع يمكن تطبيقها بالتزامن، وتتمتع بأداء يتدرج خطيًا مع مقدار التوازي.

لننفذ كل هذه العمليات بالتوازي، وبالتالي نخفي زمن وصول الملف I/O ونستخدم CPUs متعددين في حسابات تدريج الصورة. إن أولى محاولاتنا لتجربة النسخة المتزامنة تضيف وحسب كلمة go. سنتجاهل الأخطاء مؤقتًا، ونتعامل معها لاحقًا.

```
// NOTE: incorrect!
func makeThumbnails2(filenames []string) {
    for _, f := range filenames {
        go thumbnail.ImageFile(f) // NOTE: ignoring errors
    }
}
```

تعمل هذه النسخة بسرعة حقًا – أسرع من اللازم في الحقيقة نظرًا لكونها تستغرق وقتا أقل من الأصل، حتى عندما تحتوي شريحة أسماء الملف على عنصر واحد فقط. لو لم يكن هناك توازي، كيف يمكن للنسخة المتزامنة أن تعمل بشكل أسرع؟ الإجابة هي أن makeThumbnails تعود قبل إنهاء ما يجب عليها فعله. وهي تبدأ كل روتينات جو، روتين لكل اسم ملف، ولكنها لا تنتظر انتهاءهم.

لا توجد طريقة مباشرة للانتظار حتى تنتهي روتينات جو، ولكن يمكننا تغير روتين-جو الداخلي بحيث ينبّه عند اكتمال روتين-جو الخارجي من خلال إرسال حدث على قناة مشتركة. وحيث أننا نعلم أن هناك روتينات جو داخلية len(filenames) بالضبط، فإن روتين-جو الخارجي يحتاج فقط لحساب هذا العدد من الأحداث قبل عودته:

```
// makeThumbnails3 makes thumbnails of the specified files in parallel.
func makeThumbnails3(filenames []string) {
    ch := make(chan struct{})
    for _, f := range filenames {
        go func(f string) {
            thumbnail.ImageFile(f) // NOTE: ignoring errors
            ch <- struct{}{}
        }(f)
    }
    // Wait for goroutines to complete.
    for range filenames {
        <-ch
    }
}
```

لاحظ أننا مررنا القيمة f كمعطى صريح للوظيفة الجانبية بدلاً من استخدام إعلان f من تغليف الحلقة:

```
for _, f := range filenames {
    go func() {
        thumbnail.ImageFile(f) // NOTE: incorrect!
        // ...
    }()
}
```

تذكّر مشكلة متغير الحلقة المأسور داخل وظيفة مجهولة التي تحدثنا عنها في القسم 5.6.1. ستجد في الجزء أعلاه أن المتغير المنفرد f مشترك بين كل قيم الوظيفة المجهولة، ويُحدث من خلال تكرارات الحلقة المتتابعة. وبحلول الوقت الذي بدأت فيه روتينات جو الجديدة تنفيذ الوظيفة الحرفية، ستكون حلقة for حدثت f، وبدأت تكرار آخر أو انتهت تمامًا (وهو الأكثر احتمالاً)، لذا عندما تقرأ هذه روتينات جو قيمة f، سيلاحظون جميعًا أنها تمتلك قيمة العنصر الأخير في الشريحة. ومن خلال إضافة معامل صريح، سنضمن استخدامنا لقيمة f الحالية عند تنفيذ عبارة go.

ماذا لو أردنا إعادة القيم من كل روتينات جو عامل إلى روتينات جو الرئيسي؟ لو فشل استدعاء thumbnail.ImageFile في صنع ملف، فإنه سيُظهر خطأ. وستظهر النسخة التالية من makeThumbnails أول خطأ تلقته من أي عمليات التحجيم:

```
// makeThumbnails4 makes thumbnails for the specified files in parallel.
// It returns an error if any step failed.
func makeThumbnails4(filenames []string) error {
    errors := make(chan error)
    for _, f := range filenames {
        go func(f string) {
            _, err := thumbnail.ImageFile(f)
            errors <- err
        }(f)
    }
    for range filenames {
        if err := <-errors; err != nil {
            return err // NOTE: incorrect: goroutine leak!
        }
    }
    return nil
}
```

تحتوي هذه الوظيفة على عيب غير ملحوظ، عندما تقابل أول خطأ غير nil، تظهر خطأ للمستدعي، ولا تترك أي روتين-جو لاستنزاف قناة الأخطاء. إن كل روتين-جو عامل متبقي سيحجَب للأبد عندما يحاول إرسال قيمة في تلك القناة، ولن يُنهى أبدًا. هذا الموقف، تسريب روتين-جو (انظر 8.4.4)، قد يتسبب في جعل البرنامج بأكمله عالقا أو يستهلك الذاكرة بالكامل.

إن أبسط حل هو استخدام قناة صوانية ذات قدرة كافية بحيث لا يحجب أي روتين-جو عامل عندما يرسل رسالة. (الحل البديل هو إنشاء روتين-جو آخر لاستنزاف القناة بينما يعيد روتين-جو الرئيسي الخطأ الأول دون تأخير).

تستخدم النسخة التالية من makeThumbnails قناة صوانية لإعادة أسماء ملفات الصور المنتجة إضافة إلى أي أخطاء.

```
// makeThumbnails5 makes thumbnails for the specified files in parallel.
// It returns the generated file names in an arbitrary order,
// or an error if any step failed.
func makeThumbnails5(filenames []string) (thumbfiles []string, err error) {
    type item struct {
        thumbfile string
        err       error
    }
    ch := make(chan item, len(filenames))
    for _, f := range filenames {
        go func(f string) {
            var it item
            it.thumbfile, it.err = thumbnail.ImageFile(f)
            ch <- it
        }(f)
    }
    for range filenames {
        it := <-ch
        if it.err != nil {
            return nil, it.err
        }
        thumbfiles = append(thumbfiles, it.thumbfile)
    }
```

```
    return thumbfiles, nil
}
```

إن نسختنا النهائية من makeThumbnails الموضحة أدناه، تعيد إجمالي عدد البايتات التي تستهلكها الملفات الجديدة. لكن على العكس من النسخ السابقة، تستقبل أسماء الملف عبر قناة من السلاسل وليس كشريحة، حتى لا يمكننا توقع عدد تكرارات الحلقة.

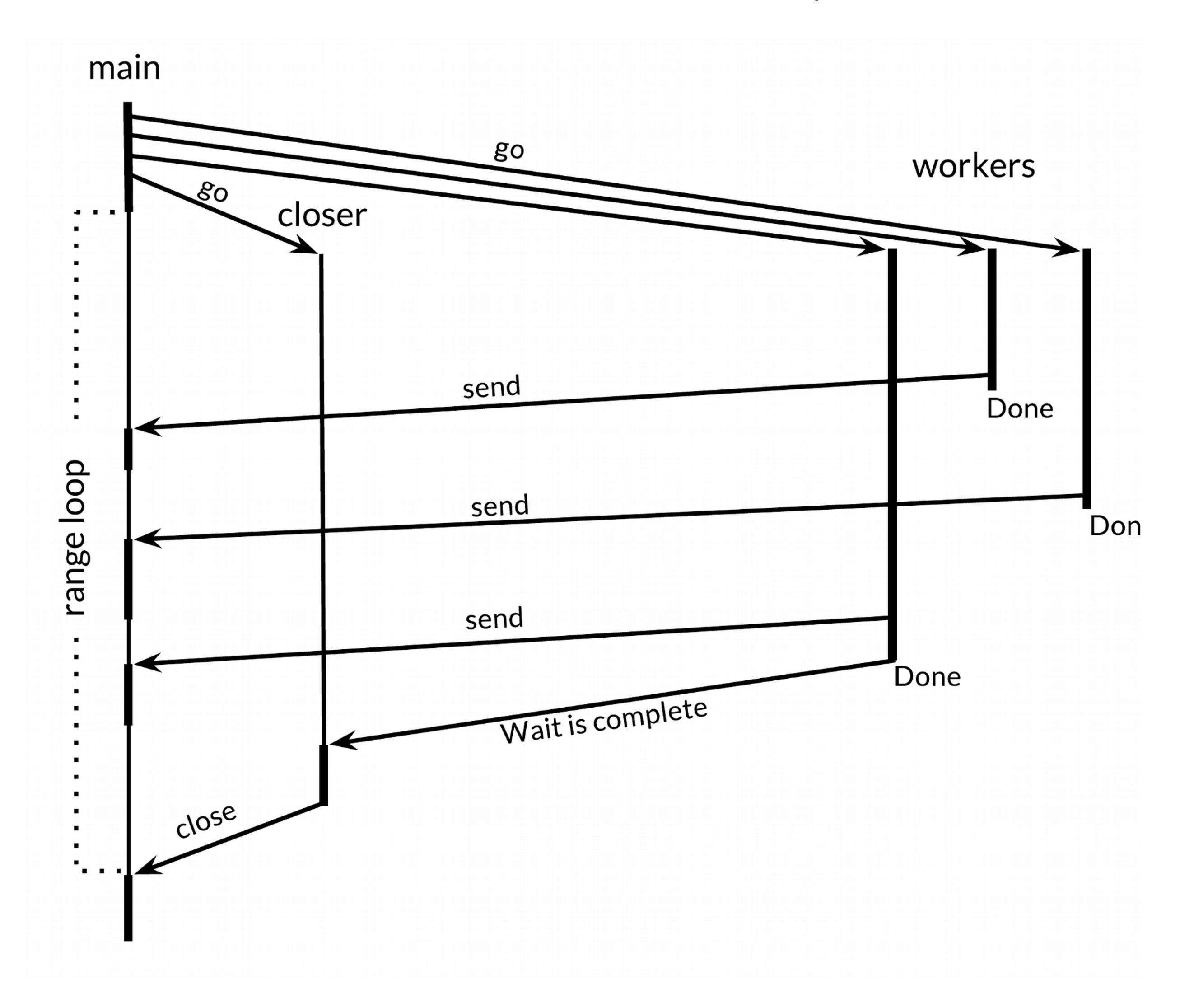
سنحتاج لزيادة العدّاد قبل بدء كل روتين-جو، وتخفيضه مع انتهاء كل روتين-جو حتى نعلم عندما ينتهي روتين-جو الأخير (والذي قد لا يكون آخر روتين-جو يبدأ). يتطلب هذا نوع مميز من العدادات، نوع يمكن التلاعب به بأمان من روتينات جو المتعددة، ويوفر طريقة انتظار حتى تصبح قيمتها صفر. إن هذا النوع من العدادات معروف sync.WaitGroup، ويوضح الكود أدناه كيف يمكن استخدامه:

```
// makeThumbnails6 makes thumbnails for each file received from the channel.
// It returns the number of bytes occupied by the files it creates.
func makeThumbnails6(filenames <-chan string) int64 {
    sizes := make(chan int64)
    var wg sync.WaitGroup // number of working goroutines
    for f := range filenames {
        wg.Add(1)
        // worker
        go func(f string) {
            defer wg.Done()
            thumb, err := thumbnail.ImageFile(f)
            if err != nil {
                log.Println(err)
                return
            }
            info, _ := os.Stat(thumb) // OK to ignore error
            sizes <- info.Size()
        }(f)
    }
    // closer
    go func() {
        wg.Wait()
        close(sizes)
    }()
    var total int64
    for size := range sizes {
        total += size
    }
    return total
}
```

لاحظ عدم التناظر في طُرق Add و Done. إن Add، التي تزيد العداد، يجب استدعاءها قبل بدء روتين-جو وليس أثناءه، وإلا لن نكون متأكدين هل حدثت Add قبل استدعاء روتين-جو "الغالِق" لـ Wait. إضافة إلى ذلك، تأخذ Add معامِل، ولكن Done لا تفعل هذا، هي أشبه بـ Add(-1). نستخدم defer لضمان أن العدّاد ينخفض حتى في حالة الخطأ. إن بنية الشفرة أعلاه هي نمط شائع واصطلاحي للتكرار الحلقي المتوازي عندما لا نعرف عدد التكرارات.

تحمل قناة sizes حجم كل ملف إلى روتين-جو الأساسي، والذي يستقبلهم باستخدام حلقة نطاق ويحسب الإجمالي. لاحظ كيف نصنع روتين-جو إغلاق ينتظر انتهاء العمال قبل إغلاق قناة sizes. يجب أن تكون هاتين العمليتين، wait و Close، متزامنتين مع الحلقة في قناة sizes. فكر في البدائل: لو كانت عملية الانتظار (wait) موضوعة في روتين-جو الرئيسي قبل الحلقة، فلن تنتهي أبدًا، ولو وُضِعت بعد الحلقة، فلن يمكن الوصول إليها حيث لن يوجد أي شيء يمكنه إغلاق القناة، ولن تنتهي الحلقة أبدًا.

يوضح الشكل 8.5 تسلسل الأحداث في وظيفة makeThumbnails6. وتمثل الخطوط الرأسية روتينات جو، بينما تمثل القطاعات الرفيعة حالة السكون والقطاعات السميكة حالة النشاط.



الشكل 8.5: تسلسل الأحداث في makeThumbnails6.

تمثل الخطوط الأفقية الرئيسية الأحداث التي تُزامِن روتين-جو مع آخر. ويتدفق الزمن لأسفل. لاحظ كيف يقضي روتين-جو الرئيسي معظم وقته في حلقة النطاق ساكنًا، في انتظار إرسال العامل لقيمة أو الغالق أن يغلق القناة.

تمرين 8.4: عدّل خادم reverb2 لاستخدام sync.WaitGroup لكل اتصال لحساب عدد روتينات جو الصدى النشطة. عندما تنخفض للصفر، اغلق نصف الكتابة في اتصال TCP كما ذكرنا في التمرين 8.3. تأكد من أن عميل netcat3 الذي عدلته في ذلك التمرين ينتظر الأصداء الأخيرة للصيحات المتزامنة المتعددة، حتى بعد إغلاق المُدخل القياسي.

تمرين 8.5: خذ برنامج تتابعي محدود بـ CPU موجود بالفعل، مثل برنامج Mandelbrot في القسم 3.3، أو حساب السطح ثلاثي الأبعاد في القسم 3.2، ونفذ حلقته الرئيسية بالتوازي مع قنوات الاتصال. ما مدى سرعة عمله على آلة متعددة المعالجات؟ ما هو عدد روتينات جو المثالي الذي يمكنك استخدامه؟

8.6 مثال: زاحف الويب المتزامن / Concurrent Web Crawler

صنعنا في القسم 5.6 زاحف ويب بسيط يستكشف الرسم البياني لرابط الويب بترتيب العرض أولا (breadth-first). وسنجعله متزامن في هذا المقال حتى تتمكن استدعاءات الزحف المستقلة من استغلال توازي I/O المتاح في الويب. ظلت وظيفة الزحف كما هي بالضبط في gopl. io/ch5/findlinks3:

```
gopl.io/ch8/crawl1

func crawl(url string) []string {
    fmt.Println(url)
    list, err := links.Extract(url)
    if err != nil {
        log.Print(err)
    }
    return list
}
```

تشبه الوظيفة الرئيسية breadthFirst (انظر 5.6)، وكما كان الحال سابقًا، تسجل قائمة عمل صف العناصر التي تحتاج للمعالجة، وكل عنصر هو قائمة URLs للزحف، ولكن هذه المرة، بدلاً من تمثيل الصف باستخدام شريحة، نستخدم قناة. يحدث كل استدعاء للزحف في روتين-جو خاص به، ويرسل الروابط التي يكتشفها إلى قائمة العمل مرة أخرى.

```
func main() {
    worklist := make(chan []string)
    // Start with the command-line arguments.
    go func() { worklist <- os.Args[1:] }()
    // Crawl the web concurrently.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                go func(link string) {
```

```
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

لاحظ أن روتين-جو crawl يأخذ link كمعامل صريح لتجنب مشكلة أسر متغير الحلقة التي شهدناها في القسم 5.6.1. لاحظ أيضًا أن الإرسال المبدئي لمعطيات سطر الأوامر إلى قائمة العمل يجب أن يعمل في روتين-جو خاص به لتجنب المأزق (deadlock)، وهو موقف عالق يحاول فيه روتين-جو الرئيسي و روتين-جو الزاحف الإرسال لبعضهما بينما لا يستقبل أي منهما. سيكون الحل البديل هو استخدام القناة الصوانية.

أصبح الزاحف عالي التزامن الآن ويطبع عاصفة من URLs، ولكن به مشكلتين. تُظهر المشكلة الأولى نفسها كرسالة خطأ في السجل بعد ثوان قليلة من بدء العملية:

```
$ go build gopl.io/ch8/crawl1
$ ./crawl1 http://gopl.io/
http://gopl.io/
https://golang.org/help/
https://golang.org/doc/
https://golang.org/blog/
...
2015/07/15 18:22:12 Get ...: dial tcp: lookup blog.golang.org: no such host
2015/07/15 18:22:12 Get ...: dial tcp 23.21.222.120:443: socket:too many open
files
...
```

إن رسالة الخطأ الأولية هي تقرير مفاجئ عن فشل بحث DNS عن نطاق موثوق فيه. أما رسالة الخطأ التالية، فتكشف عن السبب: لقد صنع البرنامج العديد من الاتصالات في وقت واحد لدرجة أنه تخطى حَد ما قبل العملية في عدد الملفات المفتوحة، مما جعل عمليات مثل بحوث DNS واستدعاءات net.Dial تبدأ في الفشل.

إن البرنامج متوازي أكثر من اللازم. ونادرًا ما تكون الموازاة غير المقيّدة فكرة جيدة حيث أن هناك دائمًا عامل مقيِّد في النظام، مثل عدد أنوية ال CPU في أحمال العمل المقيدة بالحساب، وعدد المحاور والرؤوس في عمليات القرص المحلي I/O، ومعدل نقل بيانات الشبكة الخاص بتحميلات التدفق، أو السعة الموجودة في خدمة الويب. إن الحَل هو الحَد من عدد الاستخدامات المتوازية للمورد بحيث تطابق مستوى التوازي المتاح، وأحد الطرق البسيطة لفعل هذا في مثالنا هي ضمان عدم وجود استدعاءات نشطة يزيد عددها عن "n" إلى links.Extract في وقت واحد، حيث n هي عدد أقل من حَد واصِف الملف – لنقل 20 مثلاً. تشبه هذا طريقة إدخال حارس البوابة في نادي ليلي للضيوف عندما يغادر بعض الضيوف الآخرين فقط.

يمكننا الحَد من التوازي باستخدام قناة صوانية ذات سعة n لنمذجة نموذج تزامن مبدئي يُطلق عليه "جهاز إرسال إشارات الحسابات" (counting semaphore). من الناحية المفاهيمية، تمثل كل فتحة n خالية في صوان القناة رمزا يسمح لحامله بالمتابعة. يحتاج إرسال قيمة للقناة إلى رمز، واستقبال قيمة من القناة يحرر هذا الرمز، ويصنع فتحة خالية أخرى. يضمن هذا أنه يمكن إرسال n في أغلب الحالات دون استقبال متداخل معها. (بالرغم من أنه قد يكون من البديهي أكثر معاملة الفتحات "الممتلئة" في صوان القناة كرموز، واستخدام الفتحات الخالية لتجنب الحاجة لملئ صوان القناة بعد إنشائه). نظرًا لكون نوع عنصر القناة غير مهم، سنستخدم {}struct، والتي حجمها صفر.

لنعيد كتابة وظيفة crawl حتى يمكننا وضع أقواس حول استدعاء links.Extract بواسطة العمليات للحصول على الرمز وتحريره، وبالتالي ضمان وجود 20 استدعاء نشط لها في وقت واحد. إن الحفاظ على عمليات semaphore قريبة قدر الإمكان من عملية I/O التي تنظمها هو أمر جيد.

```
gopl.io/ch8/crawl2

// tokens is a counting semaphore used to
// enforce a limit of 20 concurrent requests.
var tokens = make(chan struct{}, 20)
func crawl(url string) []string {
	fmt.Println(url)
	tokens <- struct{}{} // acquire a token
	list, err := links.Extract(url)
	<-tokens // release the token
	if err != nil {
		log.Print(err)
	}
	return list
}
```

المشكلة الثانية هي أن البرنامج لا ينتهي أبدًا، حتى بعد اكتشافه لكل الروابط الممكن الوصول لها من URLs المبدئية. (من غير المرجح بالطبع أن تلاحظ هذه المشكلة ما لم تختر URLs المبدئي بعناية، أو تطبق خاصة تقييد العمق المذكورة في التمرين 8.6). سنحتاج لكسر الحلقة الرئيسية عندما تكون قائمة العمل فارغة، وأن نضمن ألا تكون هناك أي روتينات جو زحف نشطة، حتى نتمكن من إنهاء البرنامج.

```
func main() {
	worklist := make(chan []string)
	var n int // number of pending sends to worklist
	// Start with the command-line arguments.
	n++
	go func() { worklist <- os.Args[1:] }()
	// Crawl the web concurrently.
	seen := make(map[string]bool)
	for ; n > 0; n-- {
		list := <-worklist
		for _, link := range list {
			if !seen[link] {
```

```
                seen[link] = true
                n++
                go func(link string) {
                    worklist <- crawl(link)
                }(link)
            }
        }
    }
}
```

يتتبع العداد n في هذه النسخة عدد الإرسالات التي لم تحدث بعد إلى قائمة العمل. وفي كل مرة نعرف أنه يجب إرسال عنصر إلى قائمة العمل، نزيد n، مرة قبل أن نرسل معطيات سطر الأوامر المبدئي، ومرة أخرى في كل مرة نبدأ روتين-جو زاحف. تنتهي الحلقة الرئيسية عندما تنخفض n إلى صفر، حيث لا يوجد مزيد من العمل الذي يجب القيام به.

يعمل الزاحف المتزامن الآن بسرعة أكبر بـ 20 ضِعف من زاحف العرض الموجود في القسم 5.6، وبدون أخطاء، وينتهي بشكل صحيح لو أكمل مهمته.

يوضح البرنامج أدناه حلا بديلا لمشكلة التزامن المفرط. تستخدم هذه النسخة وظيفة الزحف الأصلية التي لا تحتوي على "جهاز إرسال عَد/ counting semaphore"، ولكنها تستدعيه من بين الـ 20 روتين-جو الزحف الطويلة، وبالتالي تضمن وجود 20 طلب HTTP على الأكثر نشطين بالتزامن.

```
gopl.io/ch8/crawl3

func main() {
    worklist := make(chan []string)  // lists of URLs, may have duplicates
    unseenLinks := make(chan string) // de-duplicated URLs
    // Add command-line arguments to worklist.
    go func() { worklist <- os.Args[1:] }()
    // Create 20 crawler goroutines to fetch each unseen link.
    for i := 0; i < 20; i++ {
        go func() {
            for link := range unseenLinks {
                foundLinks := crawl(link)
                go func() { worklist <- foundLinks }()
            }
        }()
    }
    // The main goroutine de-duplicates worklist items
    // and sends the unseen ones to the crawlers.
    seen := make(map[string]bool)
    for list := range worklist {
        for _, link := range list {
            if !seen[link] {
                seen[link] = true
                unseenLinks <- link
            }
        }
    }
}
```

إن روتينات جو الزاحفة كلها تتغذى من نفس القناة unseenLinks . والـروتين-جو الرئيسي مسؤول عن عدم تكرار العناصر التي يستقبلها من قائمة العمل، ثم يرسل كل عنصر غير مرئي إلى قناة unseenLinks إلى روتين-جو زاحف.

إن خريطة seen مقيدة داخل روتين-جو الرئيسي، بمعنى، أنه يمكن الوصول لها فقط بواسطة هذا الـروتين-جو. يساعدنا التقييد – مثله مثل أنواع إخفاء المعلومات الأخرى – على التفكير بشكل منطقي في صحة البرنامج. كمثال، لا يمكننا ذكر المتغيرات المحلية بالاسم من خارج الوظيفة التي يعلنون فيها، فالمتغيرات التي لا تهرب (انظر 2.3.4) من وظيفة لا يمكن الوصول لها من خارج تلك الوظيفة، وحقول الكائن المغلفة لا يمكن الوصول إليها إلا من خلال طرق هذا الكائن. في كل الأحوال، يساعد إخفاء المعلومات على الحد من التفاعلات غير المقصودة بين أجزاء البرنامج.

إن الروابط التي يتم إيجادها من خلال الزحف تُرسل إلى قائمة العمل من روتين-جو مخصص لتجنب النهاية المغلقة.

لم نتناول مشكلة الإنهاء في هذا المثال لتوفير المساحة.

تمرين 8.6: أضِف تقييد العمق إلى الزاحف المتزامن. بمعنى، لو حدد المستخدم depth=3– ، فإن الـ URLs الوحيدة التي يمكن الوصول لها هي ثلاث روابط على الأكثر.

تمرين 8.7: اكتب برنامجا متزامنا يصنع مِرآة محلية لموقع ويب، اجلب كل صفحة يمكن الوصول لها واكتبها في دليل على القرص المحلي. يجب جلب الصفحات الموجودة في النطاق الأصلي فقط (كمثال: golang.org). يجب تعديل URLs في الصفحات المعكوسة بالمرآة حسب الحاجة بحيث تشير إلى الصفحة المعكوسة وليس الأصلية.

8.7 تعدُد الإرسال من خلال select

يقوم البرنامج أدناه بإجراء عد تنازلي لإطلاق صاروخ. وتعيد وظيفة time.Tick قناة ترسل عبرها الأحداث دوريًا، وتعمل كبندول إيقاعي. إن قيمة كل حدث هي ختم زمني، ولكنها نادرًا ما تكون مثيرة بقدر طريقة توصيلها.

```
gopl.io/ch8/countdown1

func main() {
    fmt.Println("Commencing countdown.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        <-tick
    }
    launch()
}
```

لنضيف الآن القدرة على إجهاض تسلسل الإطلاق من خلال الضغط على زر الإعادة خلال العد التنازلي. أولا، نبدأ روتين-جو يحاول قراءة بايت واحد من مُدخل قياسي، ولو نجح، فإنه يرسل قيمة على قناة اسمها abort أو إجهاض.

```
gopl.io/ch8/countdown2

abort := make(chan struct{})
go func() {
    os.Stdin.Read(make([]byte, 1)) // read a single byte
    abort <- struct{}{}
}()
```

يحتاج كل تكرار لحلقة العد التنازلي انتظار وصول الحدث في واحدة من القناتين: قناة ticker لو كان كل شيء على ما يرام (أو "اسمي" بمصطلح ناسا)، أو حدث إجهاض لو كان حالة "حالة شاذة". لا يمكننا الاستقبال من كل قناة لأن أيًا كانت العملية التي نجربها أولا، ستقوم بالحجب حتى اكتمالها. نحن نحتاج إلى إجراء "إرسال متعدد" لهذه العمليات، وسنحتاج إلى عبارة (select statement) select لفعل هذا.

```
select {
case <-ch1:
    // ...
case x := <-ch2:
    // ...use x...
case ch3 <- y:
    // ...
default:
    // ...
}
```

يظهر أعلاه الشكل العام لعبارة select. ومثل عبارة switch، تحتوي على عدد من الحالات، ووضع مبدئي اختياري. تحدد كل حالة "communication اتصال" (عملية إرسال أو استقبال في قناة ما)، وكتلة عبارات مرتبطة به. قد يظهر تعبير الاستقبال بمفرده، كما هو موضح في الحالة الأولى، أو داخل إعلان متغير قصير، كما هو موضح في الحالة الثانية، ويتيح لك الشكل الثاني الإشارة إلى القيمة المستلَمة.

تنتظر select حتى يصبح الاتصال الخاص بحالة ما جاهزا للمتابعة، ثم تقوم بأداء هذا الاتصال وتنفيذ العبارات المرتبطة بالحالة، ولا تحدث الاتصالات الأخرى. إن select التي بدون حالات، {}select، تنتظر للأبد.

لنعُد إلى برنامج إطلاق الصاروخ الخاص بها. تعيد وظيفة time.After القناة فورًا، وتبدأ روتين-جو جديد يرسل قيمة واحدة في تلك القناة بعد وقت محدد. تنتظر عبارة select أدناه حتى وصول أول حدث من ضمن حدثين، إما حدث إجهاض أو حدث يشير إلى مرور 10 ثوان. لو مرت 10 ثوان بدون إجهاض، فإن عملية الإطلاق تبدأ.

```
func main() {
    // ...create abort channel...
```

```
    //!+
    fmt.Println("Commencing countdown.  Press return to abort.")
    select {
    case <-time.After(10 * time.Second):
        // Do nothing.
    case <-abort:
        fmt.Println("Launch aborted!")
        return
    }
    launch()
}
```

إن المثال أدناه بارع أكثر. إن القناة ch، التي حجم صوانها 1، تتناوب بين حالة فراغ إلى امتلاء، بالتالي يتم متابعة حالة واحدة فقط، إما الإرسال عندما تقوم i زوجية، أو الاستقبال عندما تكون i فردية، وهي دائمًا ما تطبع: 0 2 4 6 8.

```
ch := make(chan int, 1)
for i := 0; i < 10; i++ {
     select {
     case x := <-ch:
          fmt.Println(x) // "0" "2" "4" "6" "8"
     case ch <- i:
     }
}
```

لو كان هناك حالات متعددة جاهزة، فإن select تختار حالة عشوائية، وهو ما يضمن أن كل قناة تمتلك فرصة اختيار متساوية. إن زيادة حجم الصوان في المثال السابق يجعل الناتج غير قطعي، لأنه حينما يكون الصوان لا ممتلئ ولا فارغ، فإن عبارة select تختار وكأنها تلقي عملة مجازية.

لنجعل برنامج إطلاقنا يبدأ العد التنازلي. تجعل عبارة select أدناه كل تكرار للحلقة ينتظر لمدة ثانية بانتظار الإجهاض، ولكن ليس أطول من هذا.

```
gopl.io/ch8/countdown3

func main() {
    // ...create abort channel...
    fmt.Println("Commencing countdown.  Press return to abort.")
    tick := time.Tick(1 * time.Second)
    for countdown := 10; countdown > 0; countdown-- {
        fmt.Println(countdown)
        select {
        case <-tick:
            // Do nothing.
        case <-abort:
            fmt.Println("Launch aborted!")
            return
        }
    }
    launch()
}
```

تعمل وظيفة time.Tick كما لو أنها تصنع روتين-جو يستدعي time.Sleep في حلقة، ويرسل الحدث في كل مرة يستيقظ. وعندما تعود وظيفة العد التنازلي أدناه، فإنها توقف استقبال الأحداث من tick، ولكن روتين-جو الخاص بـ tick لا يزال هناك، يحاول دون جدوى الإرسال على القناة التي لا يستقبل منها أي روتين-جو - إن الأمر هو تسرب روتين-جو (انظر 8.4.4).

إن وظيفة Tick مريحة، ولكنها تكون مناسبة فقط حينما يكون هناك حاجة لـ ticks خلال فترة عمل التطبيق كلها، بخلاف هذا، يجب أن نستخدم هذا النمط:

```
ticker := time.NewTicker(1 * time.Second)
<-ticker.C // receive from the ticker's channel
ticker.Stop() // cause the ticker's goroutine to terminate
```

قد نريد أحيانًا محاولة الإرسال أو الاستقبال على قناة ولكن مع تجنُب الحجب لو كانت القناة غير مستعدة - اتصال غير محجوب. يمكن لعبارة select أن تفعل هذا أيضا. وقد تحتوي select على وضع مبدئي يحدد ما يجب فعله عند عدم التمكن من متابعة كل الاتصالات الأخرى فورًا.

تستقبل عبارة select أدناه قيمة من قناة الإجهاض لو لم يكن هناك استقبال، وإلا فإنها لا تفعل أي شيء. إن هذه عملية استقبال غير محجوبة، وفعلها بشكل متكرر يُطلق عليه جَس (Polling) القناة.

```
select {
case <-abort:
     fmt.Printf("Launch aborted!\n")
     return
default:
     // do nothing
}
```

إن القيمة الصفرية للقناة هي nil. وربما يكون من المفاجئ أن قنوات nil قد تكون مفيدة أحيانًا. نظرًا لكون عمليات الإرسال والاستقبال في nil تُحجب للأبد، فإن الحالة في عبارة select التي قناتها nil لا يتم اختيارها أبدًا. ويجعلنا هذا نستخدم nil لإتاحة أو رفض الحالات التي تتوافق مع خصائص مثل التعامل مع الأوقات المستقطعة والإلغاء، والاستجابة لأحداث مُدخل أخرى، أو حذف مُخرج. سنرى مثال على هذا في الجزء القادم.

تمرين 8.8: باستخدام عبارة select، قم بإضافة وقت مستقطع إلى خادم الصدى في القسم 8.3، بحيث يفصل أي عميل لا يصيح بشيء خلال 10 ثوان.

8.8 مثال: اجتياز الدليل المتزامن

سنبني في هذا القسم برنامجا يقدم تقرير احول استخدام القرص بواسطة دليل أو أكثر محددين في سطر الأوامر، مثل أمر يونكس du. تقوم وظيفة walkDir أدناه بمعظم العمل، وهي تعدّد مُدخلات الدليل dir باستخدام وظيفة مساعدة dirents.

```
gopl.io/ch8/du1

// walkDir recursively walks the file tree rooted at dir
// and sends the size of each found file on fileSizes.
func walkDir(dir string, fileSizes chan<- int64) {
	for _, entry := range dirents(dir) {
		if entry.IsDir() {
			subdir := filepath.Join(dir, entry.Name())
			walkDir(subdir, fileSizes)
		} else {
			fileSizes <- entry.Size()
		}
	}
}

// dirents returns the entries of directory dir.
func dirents(dir string) []os.FileInfo {
	entries, err := ioutil.ReadDir(dir)
	if err != nil {
		fmt.Fprintf(os.Stderr, "du1: %v\n", err)
		return nil
	}
	return entries
}
```

تعيد وظيفة ioutil.ReadDir شريحة من os.FileInfo وهي نفس المعلومات التي يعيدها استدعاء لـ os.Stat لملف واحد. وبالنسبة لكل دليل فرعي، يستدعي walkDir نفسه بشكل متكرر، ويرسل walkDir رسالة في قناة fileSizes لكل ملف. إن الرسالة هم حجم الملف بالبايتات.

تستخدم الوظيفة الرئيسة الموضحة أدناه اثنين من روتينات جو. يقوم روتين-جو الخلفية باستخدام walkDir لكل دليل محدد في سطر الأوامر، ثم يغلق قناة fileSize في النهاية، بينما يقوم روتين-جو الرئيسي بحساب مجموع أحجام الملفات التي استقبلها من القناة، ثم يطبع الإجمالي في النهاية.

```
// The du1 command computes the disk usage of the files in a directory.
package main
import (
	"flag"
	"fmt"
	"io/ioutil"
	"os"
	"path/filepath"
)
```

```
func main() {
    // Determine the initial directories.
    flag.Parse()
    roots := flag.Args()
    if len(roots) == 0 {
        roots = []string{"."}
    }
    // Traverse the file tree.
    fileSizes := make(chan int64)
    go func() {
        for _, root := range roots {
            walkDir(root, fileSizes)
        }
        close(fileSizes)
    }()
    // Print the results.
    var nfiles, nbytes int64
    for size := range fileSizes {
        nfiles++
        nbytes += size
    }
    printDiskUsage(nfiles, nbytes)
}

func printDiskUsage(nfiles, nbytes int64) {
    fmt.Printf("%d files  %.1f GB\n", nfiles, float64(nbytes)/1e9)
}
```

يتوقف هذا البرنامج مؤقتًا لفترة طويلة قبل أن يطبع نتيجته:

```
$ go build gopl.io/ch8/du1
$ ./du1 $HOME /usr /bin /etc
213201 files 62.7 GB
```

سيكون البرنامج ألطف لو أبقانا على إطلاع حول تقدمه، لكن نقل استدعاء printDiskUsage إلى الحلقة ببساطة سيجعلها تطبع آلاف السطور كمُخرجات.

إن نسخة du أدناه تطبع القيم الإجمالية دوريًا، ولكن فقط لو كان عَلَم v- محدد، حيث قد لا يرغب كل المستخدمين في رؤية رسائل التقدم. إن روتين-جو الخلفية الذي يكرر roots يظل بدون تغيير، بينما يستخدم روتين-جو الرئيسي الآن خصائص (ticker) لإنتاج الأحداث كل 500 مللي ثانية، وتنتظر عبارة select إما رسالة حجم الملف، وفي تلك الحالة تقوم بتحديث الإجمالي، أو حدث خصائص (tick)، وفي تلك الحالة تطبع القيم الإجمالية الحالة. لو كان عَلَم v- غير محدد، فإن قناة tick تظل nil، وتُعطّل حالتها في select بشكل فعال.

```
gopl.io/ch8/du2

var verbose = flag.Bool("v", false, "show verbose progress messages")
func main() {
    // ...start background goroutine...
```

```
    //!-
    // Determine the initial directories.
    flag.Parse()
    roots := flag.Args()
    if len(roots) == 0 {
        roots = []string{"."}
    }
    // Traverse the file tree.
    fileSizes := make(chan int64)
    go func() {
        for _, root := range roots {
            walkDir(root, fileSizes)
        }
        close(fileSizes)
    }()
    //!+
    // Print the results periodically.
    var tick <-chan time.Time
    if *verbose {
        tick = time.Tick(500 * time.Millisecond)
    }
    var nfiles, nbytes int64
loop:
    for {
        select {
        case size, ok := <-fileSizes:
            if !ok {
                break loop // fileSizes was closed
            }
            nfiles++
            nbytes += size
        case <-tick:
            printDiskUsage(nfiles, nbytes)
        }
    }
    printDiskUsage(nfiles, nbytes) // final totals
}
```

حيث أن البرنامج لم يعد يستخدم حلقة range، فإن أول حالة select يجب أن تختبر صراحةً ما إذا كانت قناة fileSizes أغلقت أم لا، باستخدام شكل نتيجتين في عملية الاستقبال. لو أُغلقت القناة، فإن البرنامج يكسر الحلقة. وتكسر عبارة break المُصنّفة كل من select وحلقة for، بينما أن عبارة break غير المُصنّفة ستكسر select فقط، وتجعل الحلقة تبدأ التكرار التالي.

يمنحنا البرنامج الآن تيار تحديثات متمهّل:

```
$ go build gopl.io/ch8/du2
$ ./du2 -v $HOME /usr /bin /etc
28608 files 8.3 GB
54147 files 10.3 GB
93591 files 15.1 GB
127169 files 52.9 GB
175931 files 62.2 GB
```

```
213201 files 62.7 GB
```

مع ذلك، لا يزال الأمر يتطلب فترة طويلة جدًا حتى ينتهي. ولا يوجد سبب يمنع كون كل الاستدعاءات لـ walkDir تتم بالتزامن، وبالتالي يستغل التوازي في نظام القرص. إن النسخة الثالثة من du – المُقدَّمة أدناه – تصنع روتين-جو جديد لكل استدعاء لـ walkDir. وهي تستخدم sync.WaitGroup (انظر 8.5) لحساب عدد استدعاءات walkDir التي لا زالت نشطة، و روتين-جو الغالِق لإغلاق قناة fileSizes عندما ينخفض العداد للصفر.

```go
gopl.io/ch8/du3

func main() {
    // ...determine roots...
    // Traverse each root of the file tree in parallel.
    fileSizes := make(chan int64)
    var n sync.WaitGroup
    for _, root := range roots {
        n.Add(1)
        go walkDir(root, &n, fileSizes)
    }
    go func() {
        n.Wait()
        close(fileSizes)
    }()
    //!-
    // ...select loop...
}

func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    for _, entry := range dirents(dir) {
        if entry.IsDir() {
            n.Add(1)
            subdir := filepath.Join(dir, entry.Name())
            go walkDir(subdir, n, fileSizes)
        } else {
            fileSizes <- entry.Size()
        }
    }
}
```

نظرًا لكون هذا البرنامج يصنع عدة آلاف من روتينات جو في ذروته، يجب أن نغير dirents لاستخدام counting semaphore لمنعه من فتح عدد ملفات أكبر من اللازم في وقت واحد، بالضبط كما فعلنا في زاحف الويب في القسم 8.6:

```go
var sema = make(chan struct{}, 20)
// dirents returns the entries of directory dir.
func dirents(dir string) []os.FileInfo {
    sema <- struct{}{}         // acquire token
    defer func() { <-sema }() // release token
    // ...
```

تعمل هذه النسخة أسرع بعدّة أضعاف من النسخة السابقة، بالرغم من وجود الكثير من التباين من نظام إلى آخر.

تمرين 8.9: اكتب نسخة من du تحسب وتعرض دوريًا قيم إجمالية منفصلة لكل دليل من أدلة root.

8.9 الإلغاء

نحن نحتاج أحيانًا إلى توجيه روتين-جو للتوقف عما يفعله، كمثال، في خادم ويب يؤدي حساب نيابة عن عميل قد أنهى اتصاله.

لا توجد طريقة تُمكّن روتين-جو من إنهاء روتين-جو آخر بشكل مباشر، حيث أن هذا سيترك كل المتغيرات المشتركة بينهم في حالة غير محددة. كمثال، في برنامج إطلاق الصاروخ (انظر 8.7) أرسلنا قيمة واحدة إلى قناة اسمها abort، وهو ما فسره روتين-جو العد التنازلي بأنه طلب لإيقاف نفسه، ولكن ماذا لو احتجنا لإلغاء اثنين من روتينات-جو، أو عدد عشوائي منهم؟

قد يكون أحد الاحتمالات هو إرسال عدد أحداث مماثل لعدد روتينات جو التي ستلغى في قناة الإجهاض. ولو كانت بعض روتينات جو قد أنهت نفسها بالفعل، فإن العدد الذي أرسلناه سيكون أكبر من اللازم، وستصبح مرسلاتنا عالقة. من ناحية أخرى، لو أن هذه روتينات جو تفرع منها روتينات جو أخرى، فإن عددنا سيكون قليلا جدًا، وسيظل هناك بعض روتينات جو غير الواعية للإلغاء. بشكل عام، من الصعب معرفة كم عدد روتينات جو التي تعمل نيابة عنا في أي لحظة محددة. علاوة على ذلك، عندما يستقبل روتين-جو قيمة من قناة إجهاض، فإنه يستهلك تلك القيمة حتى لا تراها روتينات جو الأخرى. إن ما نحتاجه لتنفيذ الإلغاء هو آلية موثوقة لـ بث "broadcast" حدث عبر قناة بطريقة تُمكّن العديد من روتينات جو من رؤيته بينما يحدث، ويمكن أن ترى لاحقًا أنه حدث بالفعل.

تذكر أنه بعد إغلاق قناة واستُنزفت كل القيم المرسلة فيها، فإن عمليات الاستقبال اللاحقة ستتابع عملها بشكل فوري وتنتج قيمة صفرية. يمكننا استغلال هذا لإنشاء آلية بث: لا ترسل قيمة على القناة، اغلقها.

يمكننا إضافة الإلغاء إلى برنامج du الموجود في القسم السابق من خلال تغييرات قليلة بسيطة. أولا، نصنع قناة إلغاء لا تُرسل عبرها قيم أبدًا، ولكن يشير إغلاقها إلى أنه حان وقت توقف البرنامج عما يفعله. نُعرِّف أيضًا وظيفة منفعية، cancelled، تتحقق أو تجس نبض حالة الإلغاء في اللحظة التي يتم استدعاءها فيها.

```
gopl.io/ch8/du4

var done = make(chan struct{})
func cancelled() bool {
   select {
   case <-done:
```

```
        return true
    default:
        return false
    }
}
```

بعد ذلك، نقوم بصنع روتين-جو سيقرأ من مُدخَل قياسي، والذي يكون متصل بمحطة عادة، وبمجرد قراءة أي مُدخل (كمثال، ضغط المستخدم على زر الإعادة)، فإن هذا روتين-جو سيبث الإلغاء من خلال إغلاق قناة done.

```
// Cancel traversal when input is detected.
go func() {
    os.Stdin.Read(make([]byte, 1)) // read a single byte
    close(done)
}()
```

سنحتاج الآن لجعل روتينات جو الخاصة بنا تستجيب للإلغاء. سنضيف في روتين-جو الرئيسي حالة ثالثة لعبارة select تحاول الاستقبال من قناة done. تعود الوظيفة لو اختيرت هذه الحالة، ولكن قبل أن تعود، يجب أن تستنزف قناة fileSizes، وتتخلص من كل القيم حتى تُغلق القناة. تفعل هذا لضمان أن أي استدعاءات نشطة لـ walkDir يمكن أن تعمل حتى التمام دون أن تصبح عالقة في الإرسال إلى fileSizes.

```
for {
    select {
    case <-done:
        // Drain fileSizes to allow existing goroutines to finish.
        for range fileSizes {
            // Do nothing.
        }
        return
    case size, ok := <-fileSizes:
        // ...
```

يجس روتين-جو وظيفة walkDir حالة الإلغاء عندما يبدأ، ويعود دون فعل أي شيء لو كانت الحالة مستقرة. يُحوّل هذا كل روتينات جو التي صُنعت بعد الإلغاء إلى حالة "لا عمليات".

```
func walkDir(dir string, n *sync.WaitGroup, fileSizes chan<- int64) {
    defer n.Done()
    if cancelled() {
        return
    }
    for _, entry := range dirents(dir) {
        // ...
    }
}
```

قد يكون من المفيد جس حالة الإلغاء مرة أخرى داخل حلقة walkDir لتجنب صنع روتينات جو بعد حدث الإلغاء. يتضمن الإلغاء مقايضة، حيث تحتاج الاستجابة الأسرع تغييرات مقحمة أكثر على منطق البرنامج عادة. وضمان عدم

حدوث عمليات مكلفة مرة أخرى أبدًا بعد حدث الإلغاء يمكن أن يتطلب تحديث العديد من الأماكن في الشفرة الخاصة بك، ولكن يمكنك تحقيق أكبر فائدة عادة من خلال فحص الإلغاء في الأماكن القليلة المهمة.

كشف القليل من التحليل لهذا البرنامج أن عنق الزجاجة فيه كان الحصول على رمز semaphore في dirents. إن عبارة select أدناه تجعل هذه العملية قابلة للإلغاء، وتقلل تأخير الإلغاء التقليدي في البرنامج من مئات الميللي ثانية إلى عشرات فقط:

```go
func dirents(dir string) []os.FileInfo {
    select {
    case sema <- struct{}{}: // acquire token
    case <-done:
        return nil // cancelled
    }
    defer func() { <-sema }() // release token
    // ...read directory...
}
```

عندما يحدث الإلغاء، تتوقف كل روتينات جو التي في الخلفية بسرعة، وتعود الوظيفة الرئيسية. ويخرج البرنامج عند عودة الوظيفة الرئيسية بالطبع، لذا قد يكون من الصعب التمييز بين الوظيفة الرئيسية التي تنظف وراء نفسها وبين الوظيفة الرئيسية التي لا تفعل هذا. هناك خدعة مفيدة يمكننا استخدامها خلال الاختبار: بدلاً من العودة من main في حالة الإلغاء، ننفذ استدعاء panic، حينها سيقوم زمن التشغيل بتفريغ كل كومة روتينات جو في برنامجنا. ولو كان روتين-جو الرئيسي هو الوحيد الباقي، فستكون الوظيفة الرئيسية نظفت وراء نفسها، ولكن لو بقي روتينات جو أخرى، فربما لم يتم إلغاءها بشكل صحيح، أو ربما تم إلغاءها ولكن الإلغاء يحتاج لوقت، قد يستحق الأمر القليل من البحث. يحتوي تفريغ الهلع عادة على معلومات كافية للتمييز بين هذه الحالات.

تمرين 8.10: يمكن إلغاء طلبات HTTP من خلال إغلاق قناة Cancel الاختيارية في http.Request struct. عدّل زاحف الويب في القسم 8.6 لدعم الإلغاء.

تلميح: وظيفة http.Get المريحة لا تمنحك فرصة تخصيص طلب، بل بدلاً من ذلك، تقوم بصنع الطلب باستخدام http.NewRequest، وتحدد حقل Cancel الخاص به، ثم تؤدي الطلب من خلال استدعاء http.DefaultClient.Do(req).

تمرين 8.11: طبّق تنويع على برنامج fetch يطلب عدّة URLs بالتزامن بإتباع طريقة mirroredQuery الموضحة في القسم 8.4.4. وبمجرد وصول الاستجابة الأولى، قم بإلغاء الطلبات الأخرى.

8.10 مثال: خادم الدردشة

سننهي هذا الفصل بخادم الدردشة الذي يجعل العديد من المستخدمين يبثون رسائل نصية إلى بعضهم. هناك أربع أنواع روتين-جو في هذا البرنامج. وهناك مثال لكل جزء من أجزاء روتينات جو main و broadcaster، كما يوجد handleConn لكل عميل، و روتين-جو واحد لـ clientWriter. إن أداة البث هي توضيح جيد لكيفية استخدام select، حيث أنها تستجيب لثلاثة أنواع مختلفة من الرسائل.

إن وظيفة روتين-جو الرئيسي، كما هو موضح أدناه، هي الاستماع إلى وقبول اتصالات الشبكة القادمة من العملاء. وهو ينشئ لكل اتصال منها روتين-جو handleConn جديد، كما هو الحال في خادم الصدى المتزامن الذي رأيناه في بداية هذا الفصل.

```go
gopl.io/ch8/chat

func main() {
	listener, err := net.Listen("tcp", "localhost:8000")
	if err != nil {
		log.Fatal(err)
	}
	go broadcaster()
	for {
		conn, err := listener.Accept()
		if err != nil {
			log.Print(err)
			continue
		}
		go handleConn(conn)
	}
}
```

بعد ذلك، سيكون لدينا أداة البث (Broadcaster)، ويسجل متغيره المحلي clients مجموعة الحالية من العملاء المتصلين، والمعلومات الوحيدة المسجلة حول كل عميل هي هوية قناة الرسائل الصادرة منه، وسنتحدث عن هذا لاحقًا.

```go
type client chan<- string // an outgoing message channel
var (
	entering = make(chan client)
	leaving  = make(chan client)
	messages = make(chan string) // all incoming client messages
)
func broadcaster() {
	clients := make(map[client]bool) // all connected clients
	for {
		select {
		case msg := <-messages:
			// Broadcast incoming message to all
			// clients' outgoing message channels.
			for cli := range clients {
				cli <- msg
			}
		case cli := <-entering:
			clients[cli] = true
		case cli := <-leaving:
```

```
            delete(clients, cli)
            close(cli)
        }
    }
}
```

تستمع أداة البث في قنوات entering و leaving العالمية لإيجاد أي إعلانات عن وصول أو مغادرة العملاء. وعندما تستقبل أحد هذه الأحداث، فإنها تحدث مجموعة clients ، ولو كان الحدث هو مغادرة، فإنها تغلق قناة الرسائل الصادرة من العميل. تستمع أداة البث كذلك للأحداث على قناة الرسائل العالمية، والتي يرسل إليها كل عميل رسائله الواردة. عندما تستقبل أداة البث أحد هذه الأحداث، فإنها تبث الرسالة لكل عميل متصل.

لنلقي نظرة الآن على روتينات جو الخاصة بكل عميل. تصنع وظيفة handleConn قناة رسائل صادرة جديدة لعميلها، وتعلن عن وصول هذا العميل إلى أداة البحث عبر قناة الدخول. ثم تقرأ كل سطر في النص القادم من العميل، وترسل كل سطر إلى أداة البث عبر قناة الرسائل الواردة العالمية، وتسبق كل رسالة بهوية المُرسِل. عندما لا يبقى شيء يجب قراءته من العميل، تعلن handleConn عن مغادرة العميل عبر قناة المغادرة وتغلق الاتصال.

```go
func handleConn(conn net.Conn) {
    ch := make(chan string) // outgoing client messages
    go clientWriter(conn, ch)
    who := conn.RemoteAddr().String()
    ch <- "You are " + who
    messages <- who + " has arrived"
    entering <- ch
    input := bufio.NewScanner(conn)
    for input.Scan() {
        messages <- who + ": " + input.Text()
    }
    // NOTE: ignoring potential errors from input.Err()
    leaving <- ch
    messages <- who + " has left"
    conn.Close()
}

func clientWriter(conn net.Conn, ch <-chan string) {
    for msg := range ch {
        fmt.Fprintln(conn, msg) // NOTE: ignoring network errors
    }
}
```

إضافة إلى ذلك، تُنشئ وظيفة handleCon روتين-جو clientWriter لكل عميل يستقبل رسائل، بحيث يقوم بالبث لقناة الرسائل الصادرة الخاصة بالعميل ويكتبها على اتصال الشبكة الخاص بالعميل. تنتهي حلقة كاتب العميل عندما تغلق أداة البث القناة بعد استقبال إخطار المغادرة.

يوضح العرض أدناه الخادم أثناء العمل بوجود عميلين في نوافذ منفصلة على نفس الكمبيوتر، ويستخدمان netcat للدردشة:

```
$ go build gopl.io/ch8/chat
$ go build gopl.io/ch8/netcat3
$ ./chat &
$ ./netcat3
You are 127.0.0.1:64208                    $ ./netcat3
127.0.0.1:64211 has arrived                You are 127.0.0.1:64211
Hi!
127.0.0.1:64208:  Hi!                      127.0.0.1:64208:  Hi!
                                      Hi Yourself
127.0.0.1:64211: Hi yourself.                 127.0.0.1:64211: Hi yourself.
^C
                                      127.0.0.1:64208 has left
$ ./netcat3
You are 127.0.0.1:64216                    127.0.0.1:64216 has arrived
                                           Welcome.
127.0.00.1:64211: Welcome.                 127.0.0.1:64211: Welcome.
                                       ^C

127.0.0.1:64211 has left
```

أثناء استقبال جلسة دردشة لعدد n من العملاء، يشغّل هذا البرنامج عدد 2n+2 من روتينات جو المتصلة بشكل متزامن، ولكنه لا يحتاج إلى عمليات حجز صريحة (انظر 9.2) رغم ذلك. وتكون خريطة العملاء مقيدة بـ روتين-جو واحد، وهو أداة البث، وبالتالي لا يمكن الدخول إليها بشكل متزامن. إن المتغير الوحيد الذي تشترك فيها روتينات جو المتعددة هو القنوات، وأمثلة net.Conn، وكلاهما "آمن تزامنيًا". سنتحدث أكثر حول التقييد، والأمان التزامني، وتداعيات مشاركة المتغيرات عبر روتينات جو في الفصل التالي.

تمرين 8.12: اجعل أداة البث تعلن عن مجموعة حالية من العملاء لكل وافد جديد. يتطلب هذا أن تسجل مجموعة clients و قناتي entering و leaving اسم العميل أيضًا.

تمرين 8.13: اجعل خادم الدردشة يقطع اتصال العملاء الساكنين، مثل الذين لم يرسلوا رسائل خلال آخر 5 دقائق.

تلميح: استدعاء conn.Close() في روتين-جو آخر يفك حجب استدعاءات Read النشطة كما يفعل input.Scan().

تمرين 8.14: غيّر بروتوكول شبكة خادم الدردشة بحيث يقدم كل عميل اسمه عند الدخول. استخدم هذا الاسم بدلاً من عنوان الشبكة عندما تسبق كل رسالة بهوية مرسلها.

تمرين 8.15: فشل برنامج أي عميل في قراءة البيانات في الوقت المحدد يجعل كل العملاء عالقين في النهاية. عدل أداة البث لتخطي الرسائل بدلاً من الانتظار لو كان كاتب العميل غير جاهز لقبولها. أو بدلاً من ذلك، يمكنك إضافة صوان لكل قناة رسائل صادرة من العميل بحيث لا يتم إسقاط الرسائل، ويجب أن تستخدم أداة البث إرسال غير حاجب لتلك القناة.

9- التزامُن مع المتغيرات المشتركة

قدّمنا في الفصل السابق العديد من البرامج التي تستخدم روتينات جو والقنوات للتعبير عن التزامن بطريقة مباشرة وطبيعية، ولكننا مررنا سريعًا على عدد من المسائل المهمة والدقيقة التي يجب على المبرمجين وضعها في أذهانهم عند كتابة شفرة متزامنة.

سنلقي في هذا الفصل نظرة أقرب على آليات التزامن، وسنشير بشكل خاص إلى بعض المشاكل المرتبطة بمشاركة المتغيرات بين روتينات جو المتعددة، والتقنيات التحليلية لاكتشاف هذه المشكلات، وأنماط حلها. وأخيرًا، سنشرح بعض الفوارق التقنية بين روتينات جو وخيوط نظام التشغيل.

9.1 حالات التعارُض (Race Conditions)

تحدث خطوات البرنامج في البرنامج التتابعي - وهو البرنامج الذي يحتوي على روتين-جو واحد فقط - بترتيب تنفيذ مألوف يحدده منطق البرنامج. كمثال، لو كان لدينا سلسلة من العبارات، فإن الأولى تحدث قبل الثانية، وهكذا. أما في البرنامج الذي يحتوي على اثنين من روتينات جو أو أكثر، فإن الخطوات داخل كل روتين-جو تحدث بترتيب مألوف، ولكننا لا نعرف بشكل عام ما إذا كان الحدث x في روتين-جو سيحدث قبل الحدث y في روتين-جو آخر، أم سيحدث بعده، أم سيحدث بالتزامن معه. عندما لا نتمكن من القول بثقة أن حدث ما "يحدث قبل" الآخر، فإن الأحداث x و y ستُعتبر متزامنة (Concurrent).

انظر لوظيفة تعمل بطريقة صحيحة في برنامج تتابعي. هذه الوظيفة تكون "آمنة تزامنيًا" (concurrency-safe) لو استمرت في العمل بشكل صحيح حتى عند استدعاءها بشكل متزامن، أي عند استدعاءها من اثنين من روتينات جو أو أكثر، بدون مزامنة إضافية. يمكننا تعميم هذا المفهوم على مجموعة من الوظائف المتعاونة، مثل طرق وعمليات نوع محدد. يكون النوع آمن تزامنيًا لو كانت كل طرقه وعملياته التي يمكن الوصول لها آمنة تزامنيًا.

يمكننا جعل البرنامج آمن تزامنيًا بدون جعل كل نوع محدد في هذا البرنامج آمن تزامنيًا. إن الأنواع الآمنة تزامنيًا هي الاستثناء وليست القاعدة، لذا يجب أن تصل للمتغير بالتزامن فقط لو كان التوثيق الخاص بنوعه يقول أن هذا آمن. نحن نتجنب الوصول المتزامن لمعظم المتغيرات إما من خلال "تقييدهم/confining" لـ روتين-جو واحد، أو من خلال

الحفاظ على ثبات عالي المستوى لـ "الاستثناء المتبادل/mutual exclusion". سنشرح هذه المصطلحات في هذا الفصل.

على النقيض، الوظائف المُصدَّرة على مستوى الحزمة يُتوقع أن تكون آمنة تزامنيًا بشكل عام، وحيث أن المتغيرات على مستوى الحزمة لا يمكن أن تتقيد بـ روتين-جو واحد، فإن الوظائف التي تعدّلها يجب أن تنفذ الاستثناء المتبادل.

هناك العديد من الأسباب التي تجعل الوظيفة لا تعمل عند استدعاءها بالتزامن، وهذا يشمل نهاية المغلقة (deadlock) والقفل المباشر (livelock)، وضآلة الموارد (resource starvation). ليس لدينا مساحة كافية لمناقشتهم كلهم هنا، لذا سنركز على الأكثر أهمية بينهم، وهي "حالة التعارض" (race condition).

إن حالة التعارض هي الموقف الذي لا يقدّم فيه البرنامج نتيجة صحيحة لمدخلات عمليات روتينات جو المتعددة. إن حالات التعارض ضارة لأنها قد تظل كامنة في البرنامج، وتظهر بشكل متقطّع، غالبًا تحت حالات الحِمل الثقيل أو عند استخدام مُجمِّعات مختلفة، أو منصات أو بِنيات معينة. هذا يجعل من الصعب نسخها وتشخيصها.

سيكون شرح خطورة حالات التعارض أسهل عند توضيحها من خلال تشبيه الخسارة المالية، لذا سننظر لبرنامج حساب مصرفي بسيط.

```
// Package bank implements a bank with only one account.
package bank
var balance int
func Deposit(amount int) { balance = balance + amount }
func Balance() int { return balance }
```

(كان بإمكاننا كتابة جسم وظيفة الإيداع (Deposit) كـ balance += amount، وهو مكافئ لما كتبناه، ولكن الشكل الأطول سيبسِّط الشرح).

يمكننا أن نرى من نظرة واحدة في برنامج بسيط كهذا أن أي تسلسل استدعاءات لـ Deposit و Balance سيقدم الإجابة الصحيحة، أي أن Balance سيقدم مجموع كل المبالغ التي أودِعت سابقًا. ولكن لو استدعينا هذه الوظائف بالتزامن بدلاً من استدعاءها تتابعيًا، فسيكون من غير المضمون أن تقدم Balance الإجابة الصحيحة. فكر في اثنين من روتينات جو التي تمثل معاملتين في حساب بنك مشترك:

```
// Alice:
go func() {
     bank.Deposit(200)                // A1
     fmt.Println("=", bank.Balance()) // A2
}()

// Bob:
go bank.Deposit(100)                  // B
```

أودعت أليس 200$ ثم فحصت رصيدها، بينما قام بوب بإيداع 100$، وحيث أن الخطوات A1 و A2 حدثتا بالتزامن مع B، لا يمكننا توقع الترتيب الذي حدثت به. قد يبدو من البديهي أن هناك ثلاث ترتيبات محتملة فقط، وسنطلق عليها "أليس أولا"، "بوب أولا"، و"أليس/بوب/أليس". يوضح الجدول التالي قيمة متغير الرصيد balance بعد كل خطوة. تمثل السلاسل المقتبسة احتمالات الرصيد المطبوع.

```
أليس أولا                 بوب أولا                       أليس/بوب/أليس
          0                                0                              0
A1    200                 B          100                 A1          200
A2  " = 200"              A1         300                 B           300
B      300                A2  " =    300"                A2  " =   300"
```

سيكون الرصيد النهائي في كل الأحوال هو 300$. الاختلاف الوحيد هو ما إذا كان رصيد أليس يتضمن المعاملة التي أجراها بوب أم لا، ولكن كلا العميلين سيشعران بالرضا في كل الأحوال.

لكن هذا التخمين البديهي خاطئ. هناك نتيجة رابعة محتملة، وفيها يحدث إيداع بوب في وسط إيداع أليس، بعد قراءة الرصيد (balance + amount)، ولكن قبل تحديثه (balance = ...)، مما سيجعل معاملة بوب تختفي. هذا لأن عملية إيداع أليس A1، هي فعليًا تسلسل من عمليتين، قراءة وكتابة، سمّيهم A1 r و A1w. وهذا هو الإدخال الإشكالي:

```
تعارض البيانات

A1r     0       ... = balance + amount
B      100
A1w         200      balance = ...
A2   "= 200"
```

بعد A1r، يقيّم التعبير balance + amount ما يصل إلى 200، لذا تكون هذه هي القيمة المكتوبة خلال A1w، بالرغم من الإيداع الذي حدث في الوسط. سيكون الرصيد النهائي 100$ فقط، وسيصبح البنك أغنى ب 100$ على حساب بوب.

يحتوي هذا البرنامج على نوع معين من حالة التعارض اسمها تعارض البيانات (data race). يحدث تعارض البيانات كلما دخل اثنين من روتينات جو إلى نفس المتغير بالتزامن، وكان أحد الإدخالات على الأقل هو write.

تصبح الأمور أكثر فوضوية لو كان تعارض البيانات يتضمن متغير من نوع أكبر من كلمة آلية واحدة، مثل واجهة أو سلسلة أو شريحة. تُحدّث هذه الشفرة x بالتزامن لشريحتين مختلفيّ الطوال:

```
var x []int
go func() { x = make([]int, 10) }()
go func() { x = make([]int, 1000000) }()
x[999999] = 1 // NOTE: undefined behavior; memory corruption possible!
```

إن قيمة x في العبارة الأخيرة غير مُحددة، ويمكن أن تكون صفرا، أو قد تكون شريحة طولها 10، أو شريحة طولها 1000000. ولكن تذكر أن هناك ثلاث أجزاء في الشريحة: المؤشر، والطول، والسعة. لو أتى المؤشر من أول استدعاء لـ make، وأتى الطول من الاستدعاء الثاني، فإن x ستكون chimera، وهي شريحة طولها الاسمي 1000000، ولكن مصفوفتها الضمنية بها 10 عناصر فقط. وفي هذه الحالة، فإن تخزين العنصر 999999 سيضيف بقوة موقع ذاكرة عشوائي بعيد، ويؤدي لعواقب من المستحيل توقُّعها، ومن الصعب الكشف عن الأخطاء فيها وتحديد موضعها. إن حقل الألغام الدلالي هذا يُطلق عليه "السلوك غير المُعرّف" (undefined behavior)، وهو معروف بين مبرمجي لغة C، ولحسن الحظ نادرًا ما تحدث مشكلات في لغة Go كما تحدث في C.

إن المفهوم القائل بأن البرنامج المتزامن هو إدخال من برامج تتابعية متعددة هو أيضًا تخمين خاطئ. كما سنرى في القسم 9.4، قد تقدّم تعارضات البيانات نتائج أكثر غرابة أيضًا. هناك العديد من المبرمجين - بعضهم شديد البراعة - سيقدم من حين لآخر تبريرات لتعارضات البيانات المعروفة في برامجهم: "تكلفة الاستثناء المتبادل مرتفعة جدًا"، "هذا المنطق لأجل رسائل الخطأ فقط"، "لا أمانع لو أسقطت بعض الرسائل"، إلخ. إن غياب المشكلات في مترجم أو مُجمّع أو منصة معينين قد يمنحهم ثقة زائفة. وهناك قاعدة عامة جيدة تقول أن "لا يوجد شيء اسمه تعارض بيانات حميد". كيف نتجنب تعارض البيانات في برامجنا إذًا؟

سنكرّر التعريف نظرًا لأهميته: يحدث تعارض البيانات عندما يصل اثنان من روتينات جو إلى متغير في نفس الوقت بالتزامن، وكان أحد حالات الدخول هذه على الأقل هو "write" أو كتابة. نرى من هذا التعريف أن هناك ثلاث طرق لتجنب تعارض البيانات.

الطريقة الأولى هي عدم كتابة المتغير. انظر للخريطة أدناه، وهي خريطة منخفضة الازدحام حيث أن كل مفتاح يُطلب لأول مرة فقط. لو تم استدعاء Icon تتابعيًا، فإن البرنامج سيعمل بشكل طبيعي، ولكن لو تم استدعاء Icon تزامنيًا، فسيكون هناك تعارض في البيانات الداخلة للخريطة.

```
var icons = make(map[string]image.Image)

func loadIcon(name string) image.Image

// NOTE: not concurrency-safe!
func Icon(name string) image.Image {
    icon, ok := icons[name]
    if !ok {
        icon = loadIcon(name)
        icons[name] = icon
    }
    return icon
}
```

لو قمنا بدلاً من هذا بتهيئة الخريطة بكل المُدخلات الضرورية قبل صُنع روتينات جو إضافية، ولم نعدلها مرة أخرى أبدًا، فإن أي عدد من روتينات جو يمكنه استدعاء Icon تزامنيًا بأمان حيث أن كل منهم سيقوم بقراءة الخريطة فقط.

```
var icons = map[string]image.Image{
     "spades.png":    loadIcon("spades.png"),
     "hearts.png":    loadIcon("hearts.png"),
     "diamonds.png": loadIcon("diamonds.png"),
     "clubs.png":     loadIcon("clubs.png"),
}
// Concurrency-safe.
func Icon(name string) image.Image { return icons[name] }
```

يُقدّم متغير لأيقونات (icons) في المثال أعلاه خلال تهيئة الحزمة، وهو ما يحدث قبل بدء تشغيل الوظيفة الرئيسية للبرنامج. لا تُعدّل الأيقونات أبدًا بمجرد تهيئتها. إن بِنيات البيانات التي لا تُعدل أبدًا أو الثابتة تعد آمنة تزامنيًا بطبيعتها، ولا يوجد حاجة للمزامنة فيها. مع ذلك، من الواضح أننا لا نستطيع استخدام هذه الطريقة لو كانت التحديثات ضرورية، كما هو الحال في الحساب المصرفي.

الطريقة الثانية لتجنب تعارض البيانات هي تجنب الوصول الدخول للمتغير من روتينات جو متعددة. هذه هي الطريقة التي تعمل بها العديد من البرامج المُقدّمة في الفصل السابق. كمثال، الـ روتين-جو الرئيسي في زاحف الويب المتزامن (انظر 8.6) هو روتين-جو منفرد يدخل للخريطة seen، و روتين-جو البث broadcaster في خادم الدردشة (انظر 8.10) هو روتين-جو الوحيد الذي يدخل لخريطة العملاء. هذه المتغيرات مُقيّدة بـ روتين-جو واحد.

ونظرًا لأن روتينات جو الأخرى لا يمكنها الوصول إلى المتغير بشكل مباشر، يجب أن تستخدم قناة لإرسال طلب استفسار أو تحديث للمتغير إلى روتين-جو المُقيِّد. هذا هو المقصود بشعار Go "لا تتواصل من خلال مشاركة الذاكرة، ولكن شارك الذاكرة من خلال التواصل". إن روتين-جو الذي يعمل كوسيط للوصول إلى متغير مُقيّد باستخدام طلبات القناة يُطلق عليه " روتين- جو المراقبة" (monitor goroutine) الخاص بهذا المتغير. كمثال، روتين-جو broadcaster يراقب الدخول لخريطة العملاء.

إليك مثال البنك وقد أعدنا كتابته مع تقييد متغير balance بـ روتين-جو مراقبة يُطلق عليه الصرّاف أو teller:

```
gopl.io/ch9/bank1
// Package bank provides a concurrency-safe bank with one account.
package bank
var deposits = make(chan int) // send amount to deposit
var balances = make(chan int) // receive balance
func Deposit(amount int) { deposits <- amount }
func Balance() int        { return <-balances }
func teller() {
   var balance int // balance is confined to teller goroutine
   for {
```

```
        select {
        case amount := <-deposits:
            balance += amount
        case balances <- balance:
        }
    }
}
func init() {
    go teller() // start the monitor goroutine
}
```

إذا لم نستطع تقييد متغير بـ روتين-جو واحد طوال عمره، فإن التقييد سيظل حلاً لمشكلة الدخول المتزامن بالرغم من ذلك. كمثال، من الشائع مشاركة متغير بين روتينات جو أثناء عملية التوارُد من خلال تمرير عنوانه من مرحلة إلى المرحلة التالية عبر قناة. لو كانت كل مرحلة في التوارد تمتنع عن دخول المتغير بعد إرساله إلى المرحلة التالية، فإن كل محاولات الدخول للمتغير ستكون تتابعيه. وفي الواقع، سيصبح المتغير مقيّد بمرحلة واحدة في التوارد، ثم يصبح مقيّد بالمرحلة التالية، إلخ. يُطلق على هذا النظام أحيانًا "التقييد المتسلسل" (Serial confinement).

إن مثال الكعكات (cakes) مقيد تسلسليًا في المثال أدناه، أولاً ب روتين-جو الخبّاز (baker)، ثم بـ روتين-جو المُزيّن (icer):

```
type Cake struct{ state string }

func baker(cooked chan<- *Cake) {
	for {
		cake := new(Cake)
		cake.state = "cooked"
		cooked <- cake // baker never touches this cake again
	}
}

func icer(iced chan<- *Cake, cooked <-chan *Cake) {
	for cake := range cooked {
		cake.state = "iced"
		iced <- cake // icer never touches this cake again
	}
}
```

الطريقة الثالثة لتجنُب تعارُض البيانات هي السماح بالعديد من روتينات جو بدخول المتغير، ولكن واحد فقط في المرة الواحدة. تُعرف هذه الطريقة بـ "الاستثناء المتبادل" (mutual exclusion)، وهو موضوع الجزء التالي.

التمرين 9.1: أضف وظيفة Withdraw(amount int) bool إلى برنامج gopl.io/ch9/bank1. يجب أن توضح النتيجة ما إذا كانت المعاملة نجحت أو فشلت بسبب الأموال غير الكافية. ويجب أن تحتوي الرسالة المرسلة إلى روتين-جو

المراقبة على كل من مبلغ السحب والقناة الجديدة التي يمكن لـ روتين-جو المراقبة إرسال النتيجة المنطقية عبرها مرة أخرى إلى Withdraw.

9.2: الاستثناء المتبادل: sync.Mutex

استخدمنا في القسم 8.6 قناة صوانية كـ "جهاز إشارات عَد" أو (Counting semaphore) لضمان عدم قيام أكثر من 20 روتينات جو بتقديم طلبات HTTP متزامنة. وباستخدام نفس الفكرة، يمكننا استخدام قناة سِعتها 1 لضمان دخول روتين-جو واحد فقط على الأكثر لمتغير مشترك في المرة الواحد. إن السيمافور الذي يعد حتى 1 فقط يُطلق عليه "السيمافور الثنائي" (binary semaphore).

```
gopl.io/ch9/bank2
var (
    sema    = make(chan struct{}, 1) // a binary semaphore guarding balance
    balance int
)
func Deposit(amount int) {
    sema <- struct{}{} // acquire token
    balance = balance + amount
    <-sema // release token
}
func Balance() int {
    sema <- struct{}{} // acquire token
    b := balance
    <-sema // release token
    return b
}
```

إن نمط "الاستثناء المتبادل" هذا مفيد جدًا لأنه مدعوم مباشرة من نوع Mutex من حزمة sync. وتستحوذ طريقة Lock على رمز (اسمه lock)، وطريقة Unlock فيه تحرر هذا الرمز:

```
gopl.io/ch9/bank3
import "sync"
var (
    mu      sync.Mutex // guards balance
    balance int
)
func Deposit(amount int) {
    mu.Lock()
    balance = balance + amount
    mu.Unlock()
}
func Balance() int {
    mu.Lock()
    b := balance
    mu.Unlock()
```

```
    return b
}
```

كل مرة يدخل روتين-جو إلى متغيرات البنك (وهي balance فقط هنا)، يجب أن يستدعي طريقة Lock في mutex للحصول على قفل حصري. لو حصل روتين-جو آخر على القفل، فإن هذه العملية ستُحظَر حتى يستدعي الـ روتين-جو الآخر Unlock، ويصبح القفل متاح مرة أخرى. يحمي mutex المتغيرات المشتركة. بالتبعية، فإن المتغيرات التي يحميها mutex تُعلن فورًا بعد إعلان mutex نفسه. لو انحرفت عن هذا، تأكد من توثيق ما فعلت.

إن منطقة الشفرة بين Lock و Unlock التي يكون روتين-جو حرا في قراءة وتعديل المتغيرات المشتركة فيها يُطلق عليها "القسم الحرج" (Critical Section). ويحدث استدعاء حامل القفل لـ Unlock قبل أن يتمكن أي روتين-جو آخر من الحصول على القفل لنفسه. من الجوهري أن يحرر روتين-جو القفل بمجرد انتهاءه، في كل مسارات الوظيفة، بما في ذلك مسارات الخطأ.

يقدم برنامج البنك أعلاه مثالاً على نمط التزامن الشائع. مجموعة من الوظائف المُصدّرة التي تغلّف متغير أو أكثر بحيث تصبح هذه الوظائف هي الطريقة الوحيدة لدخول هذه المتغيرات (أو يمكن دخولها عبر الطُرق، لو كانت المتغيرات متغيرات كائن). تكتسب كل وظيفة قفل mutex في البداية وتحرره في النهاية، وبالتالي تضمن عدم الدخول للمتغيرات المشتركة بالتزامن. إن هذا الترتيب للوظائف، وقفل mutex والمتغيرات يُطلق عليه "المراقبة" (Monitor). (إن استخدام كلمة مراقبة القديم monitor هذا قد ألهم باستخدام مصطلح "monitor goroutine". يشترك كلا الاستخدامين في معنى الوسيط الذي يضمن دخول المتغيرات تتابعيًا).

نظرًا لكون الأقسام الحرجة في وظائف Deposit و Balance قصيرة جدًا – سطر واحد، لا أقواس – فإن استدعاء Unlock في النهاية سيكون مباشر. أما في الأقسام الحرجة الأكثر تعقيدًا، وخاصة تلك التي يجب التعامل مع الأخطار فيها من خلال الإعادة مبكرًا، فقد يكون من الصعب معرفة ما إذا كانت استدعاءات Lock و Unlock مقترنة معًا بشكل صارم في كل المسارات. تأتي عبارة defer في لغة Go لإنقاذ الوضع، فمن خلال إرجاء الاستدعاء لـ Unlock، يمتد القسم الحرج ضمنيًا إلى نهاية الوظيفة الحالية، ويحررنا من الاضطرار لتذكُر إدخال استدعاءات Unlock في مكان أو أكثر بعيدًا عن استدعاء Lock.

```
func Balance() int {
    mu.Lock()
    defer mu.Unlock()
    return balance
}
```

نجد في المثال أعلاه أن Unlock تنفّذ "بعد" أن تقرأ عبارة return قيمة الرصيد، بالتالي تكون وظيفة Balance آمنة تزامنيًا. إضافة إلى هذا، لم نعد بحاجة للمتغير المحلي b.

علاوة على ذلك، ستعمل Unlock المؤجلة حتى لو حدث هلعا بالقسم الحرج، وهو ما قد يكون مهما في البرامج التي تستخدم recover (انظر 5.10). إن الإرجاء أو التأجيل defer سيكون مكلفا أكثر بنسبة طفيفة من الاستدعاء الصريح لـ Unlock، ولكنه ليس مكلفا بما يكفي لتبرير تقديم شفرة أقل وضوحًا. وكما هو الحال دائمًا في البرامج المتزامنة، فضِّل الوضوح وقاوم التحسين السابق لأوانه. استخدم الإرجاء كلما أمكن، واترك الأقسام الحرجة تمتد حتى نهاية الوظيفة.

انظر لوظيفة Withdraw أدناه. لو نجحت، ستقلل الرصيد بالمبلغ المحدد وتعيد true، ولكن لو كان هناك مبلغ غير كافي للمعاملة في الحساب، فإن Withdraw تستعيد الرصيد وتعيد false.

```
// NOTE: not atomic!
func Withdraw(amount int) bool {
	Deposit(-amount)
	if Balance() < 0 {
		Deposit(amount)
		return false // insufficient funds
	}
	return true
}
```

تقدم هذه الوظيفة في النهاية الإجابة الصحيحة، ولكن لها تأثير جانبي سيء جدًا. عند محاولة القيام بسحب مبالغ فيه، ينخفض الرصيد مؤقتًا لأقل من الصفر، وقد يسبب هذا رفض السحب المتزامن لمبلغ متواضع. لذا لو حاول بوب شراء سيارة رياضية، فإن أليس لن تتمكن من شراء قهوتها الصباحية. إن المشكلة هي أن Withdraw ليست ذرية "atomic"، بل تتكوّن من سلسلة من ثلاث وظائف منفصلة، كل منها يكتسب ثم يحرر قفل mutex، ولكن لا يوجد شيء يُغلق التسلسل بأكمله.

إن الحالة المثالية هي أن Withdraw ستحصل على قفل mutex مرة واحدة خلال العملية بأكملها، ولكن هذه المحاولة لن تنجح:

```
// NOTE: incorrect!
func Withdraw(amount int) bool {
	mu.Lock()
	defer mu.Unlock()
	Deposit(-amount)
	if Balance() < 0 {
		Deposit(amount)
		return false // insufficient funds
	}
	return true
}
```

تحاول Deposit الحصول على قفل mutex للمرة الثانية من خلال استدعاء ()mu.Lock، ولكن لأن أقفال mutex ليست "متعددة الدخول" (re-entrant) – سيكون من غير الممكن إقفال mutex مُقفَل بالفعل- سيؤدي هذا إلى مأزق حيث يتوقف كل شيء ولا يمكننا المتابعة، وتصبح Withdraw محظورة للأبد.

هناك سبب جيد يجعل mutexes لغة Go غير متعددة الدخول، الهدف من الـ mutex هو ضمان أن هناك ثوابت معينة في المتغيرات المشتركة موجودة في النقاط الحرجة خلال تنفيذ البرنامج، وأحد هذه الثوابت هو "عدم دخول روتين-جو للمتغيرات المشتركة"، ولكن قد يكون هناك ثوابت أخرى إضافية مخصصة لبنيات البيانات التي تحميها mutex. عندما يحصل روتين-جو على قفل mutex، فقد يفترض أن الثوابت سارية، وبينما يسيطر على القفل، قد يقوم بتحديث المتغيرات المشتركة بحيث يتم انتهاك الثوابت مؤقتًا. مع ذلك، عندما يُحرر القفل، يجب أن يضمن استعادة النظام، وأن الثوابت أصبحت سارية مرة أخرى. وبالرغم من أن mutex متعدد الدخول سيضمن عدم دخول روتينات جو أخرى للمتغيرات المشتركة، إلا أنه لن يتمكن من حماية الثوابت الإضافية من هذه المتغيرات.

إن الحل الشائع هو تقسيم وظيفة مثل Deposit إلى اثنين: وظيفة غير مُصدّرة، deposit، تفترض أن القفل يعمل بالفعل ويقوم بالعمل الحقيقي، ووظيفة مصّدرة، Deposit، تحصل على القفل قبل استدعاء deposit. يمكننا التعبير حينها عن withdraw من حيث deposit كالتالي:

```
func Withdraw(amount int) bool {
     mu.Lock()
     defer mu.Unlock()
     deposit(-amount)
     if balance < 0 {
           deposit(amount)
           return false // insufficient funds
     }
     return true
}

func Deposit(amount int) {
     mu.Lock()
     defer mu.Unlock()
     deposit(amount)
}

func Balance() int {
     mu.Lock()
     defer mu.Unlock()
     return balance
}
// This function requires that the lock be held.
func deposit(amount int) { balance += amount }
```

إن وظيفة deposit الموضحة هنا تافهة جدًا بالطبع لدرجة أن وظيفة Withdraw الواقعية لن تزعج نفسها باستدعائها، ولكنها توضح المبدأ رغم هذا.

إن التغليف (انظر 6.6)، الذي يتم من خلال تقليل التفاعلات غير المتوقعة في البرنامج، يساعدنا على الحفاظ على ثوابت بنية البيانات. ولنفس السبب، يساعدنا التغليف كذلك على الحفاظ على ثوابت التزامن. عندما تستخدم mutex، تأكد من أن mutex والمتغيرات التي يحميها غير مُصدّرين، سواء كانت متغيرات على مستوى الحزمة أو حقول struct.

9.3 Mutexes القراءة/الكتابة: sync.RWMutex

مَر بوب بنوبة قلق بعد أن شاهد إيداعه البالغ 100$ يختفي دون أثر، فكتب برنامج يتحقق من رصيده في البنك مئات المرات في الثانية، وشغّل البرنامج في المنزل وفي العمل وعلى الهاتف. لاحظ البنك أن المرور المتزايد يؤخر الإيداعات والمسحوبات، لأن كل طلبات الرصيد تعمل بالتزامن، ويتمسك بالقفل بشكل حصري، ويمنع روتينات جو الأخرى من العمل مؤقتًا.

نظرًا لكون وظيفة Balance تحتاج فقط لـ "قراءة" حالة المتغير، فسيكون من الآمن مضاعفة استدعاءات Balance لتعمل بالتزامن، طالما أنه لا يوجد استدعاء Deposit أو Withdraw يعمل معها. سنحتاج في هذا السيناريو إلى نوع خاص من الأقفال التي تسمح بمتابعة عمليات القراءة فقط بالتوازي مع بعضها، ولكن يجب أن تكون عمليات الكتابة ذات دخول استثنائي بالكامل. يُطلق على هذا القفل قِفل "قراء متعددين، كاتب واحد" (multiple readers, single writer)، وهو يُقدّم بواسطة sync.RWMutex في لغة Go:

```
var mu sync.RWMutex
var balance int

func Balance() int {
     mu.RLock() // readers lock
     defer mu.RUnlock()
     return balance
}
```

تستدعي وظيفة Balance الآن طُرق RLock و RUnlock لاكتساب وتحرير قفل readers و shared. إن وظيفة Deposit التي لم تتغير، تستدعي طُرق mu.Lock و mu.Unlock لاكتساب وتحرير قفل writer أو exclusive.

بعد هذا التغيير، ستعمل كل طلبات Balance الخاصة ببوب على التوازي مع بعضها، وتنتهي بشكل أسرع. سيصبح القفل متاح لوقت أطول، ويمكن متابعة طلبات Deposit في الوقت المحدد.

يمكن استخدام RLock فقط لو لم يكن هناك كتابة على المتغيرات المشتركة في القسم الحرج. ولا يجب أن نفترض عامةً أن وظائف أو طُرق القراءة فقط المنطقية لا تُحدّث بعض المتغيرات. كمثال، الطريقة التي تبدو مدخل بسيطة يمكن أن تزيد استخدام العداد الداخلي، أو تُحدِّث المخبأ حتى تردد الاستدعاءات أسرع، ولو كنت تشعر بالشكل فاستخدم Lock حصري.

سيكون من المُربح استخدام RWMutex فقط عندما تكون معظم روتينات جو التي تحصل على القفل قُرّاء، والقفل محل تنازع، بمعنى أن روتينات جو يجب أن تنتظر حتى تحصل عليه. تتطلب RWMutex حفظ سجلات داخلي أكثر تعقيدًا، مما يجعلها أبطأ من mutex عادي للأقفال غير المُتنازَع عليها.

9.4 مزامنة الذاكرة

قد تتساءل لماذا تحتاج طريقة Balance إلى استثناء متبادل، القائم على القناة أو القائم على الـ mutex. فبعد كل شيء، وعلى العكس من Deposit، تتكون Balance من عملية واحدة، وبالتالي لا يوجد خطر ناتج عن قيام روتين-جو آخر بالتنفيذ "في أثناء تشغيلها". نحن نحتاج mutex لسببين. الأول هو أنه من المهم ألا يتم تنفيذ Balance في وسط عملية أخرى مثل Withdraw، والسبب الثاني (والأدق) هو أن المزامنة تدور حول ما هو أكثر من ترتيب تنفيذ روتينات جو متعددة، فالمزامنة تؤثر على الذاكرة أيضًا.

قد يكون هناك العشرات من المعالِجات في أجهزة الكمبيوتر الحديثة، كل منها له مخبأه المحلي الخاص في الذاكرة الرئيسية. إن الكتابات على الذاكرة تُصان داخل كل معالِج وتُخرج للذاكرة الرئيسية فقط عند الضرورة من أجل تحقيق المزيد من الكفاءة. وقد تُدخل في الذاكرة الرئيسية بترتيب مختلف عن الترتيب الذي كُتبت به بواسطة روتين-جو الكتابة. إن أساسيات المزامنة، مثل اتصالات القناة وعمليات mutex، تجعلان المعالج يخرج محتوياته ويودِع كل كتاباته المتراكمة بحيث يضمن أن تكون آثار تنفيذ روتين-جو حتى هذه النقطة واضحة لروتينات جو التي تعمل في المعالِجات الأخرى.

انظر للمخرجات المحتملة لجزء الشفرة التالي:

```
var x, y int
go func() {
    x = 1                           //   A1
    fmt.Print("y:", y,   "    ")    //   A2
}()
go func() {
    y = 1                           //   B1
    fmt.Print("x:", x,   "    ")    //   B2
```

```
}()
```

حيث أن روتينيّ جو هذين متزامنين ويدخلان المتغيرات المشتركة بدون استثناء متبادل، فسنجد أن هناك تعارض بيانات، وبالتالي لا يجب أن نتفاجئ أن البرنامج غير حاسم. ويجب أن نتوقع أن يطبع أي من هذه النتائج الأربعة، والتي تتوافق مع المدخلات البديهية للعبارات المصنفة في البرنامج:

```
y:0 x:1
x:0 y:1
x:1 y:1
y:1 x:1
```

يجب تفسير السطر الرابع بالتسلسل: A1,B1,A2,B2 أو B1,A1,A2,B2. ولكن هذين النتيجتين قد تكونا مفاجأة:

```
x:0 y:0
y:0 x:0
```

ولكن بناء على المترجِم، والـ CPU، والعديد من العوامل الأخرى، يمكن أن يحدثا أيضًا. ما هو الإدخال المحتمل الذي يمكنه تفسير العبارات الأربعة؟

إن آثار كل عبارة داخل كل روتين-جو منفرد من المؤكد أن تحدث بترتيب تنفيذها، لأن روتينات جو متسقة تتابعيًا. ولكن في غياب التزامن الصريح باستخدام قناة أو mutex، لا يوجد ضمان أن كل روتينات جو سترى الأحداث بنفس الترتيب. بالرغم من أن روتين-جو (A) يجب أن ينتبه لتأثير كتابة x = 1 قبل قراءة قيمة y، إلا أنه قد لا ينتبه بالضرورة للكتابة على y التي قام بها الـ روتين-جو (B)، بالتالي يمكن أن يطبق A قيمة قديمة لـ y.

سنشعر بإغراء محاولة فهم التزامن كما لو أنه يستجيب "لبعض" إدخالات عبارات كل روتين-جو، ولكن كما يوضح المثال أعلاه، ليست هذه هي الطريقة التي يعمل بها المترجم الحديث أو الـ CPU. تشير المهمة و Print إلى متغيرات مختلفة، وبالتالي يمكن للمترجم استنتاج أن ترتيب عبارتين لا يؤثر على النتيجة ويبدّل بينهما. لو تم تنفيذ روتينيّ روتينات جو في CPUs مختلفة، كل منها بمخبأه الخاص، فإن الكتابات الخاصة بـ روتين-جو منهم لن تكون ظاهرة لـ Print الـ روتين-جو الآخر حتى تتزامن المخابئ داخل الذاكرة الرئيسية.

يمكن تجنب كل مشكلات التزامن هذه من خلال الاستخدام المتسق لأنماط بسيطة وقائمة. وكلما أمكن، قيِّد المتغيرات بـ روتين-جو واحد، واستخدم الاستثناء المتبادل مع كل المتغيرات الأخرى.

9.5 التهيئة الكسولة: `sync.Once`

إن إجراء خطوة التهيئة المكلفة حتى لحظة الاحتياج لها هو أحد الممارسات الجيدة، فتهيئة متغير مقدمًا يزيد زمن الوصول في بدء البرنامج، وهو غير ضروري لو لم يصل التنفيذ دائمًا للجزء الذي يستخدم المتغير في البرنامج. لتعُد إلى متغير الأيقونات الذي رأيناه سابقًا في هذا الفصل.

```
var icons map[string]image.Image
```

تستخدم هذه النسخة من Icons التهيئة الكسولة:

```
func loadIcons() {
	icons = map[string]image.Image{
		"spades.png":   loadIcon("spades.png"),
		"hearts.png":   loadIcon("hearts.png"),
		"diamonds.png": loadIcon("diamonds.png"),
		"clubs.png":    loadIcon("clubs.png"),
	}
}
// NOTE: not concurrency-safe!
func Icon(name string) image.Image {
	if icons == nil {
		loadIcons() // one-time initialization
	}
	return icons[name]
}
```

يمكننا أن نستخدم النمط أعلاه مع المتغير الذي يدخله روتين-جو منفرد فقط، ولكن هذا النمط لن يكون آمنا لو تم استدعاء Icon بالتزامن. وكما هو الحال في وظيفة Deposit الأصلية الخاصة بالبنك، يتكون Icon من خطوات متعددة: يختبر ما إذا كانت الأيقونات قيمتها صفر، ثم يحمّل الأيقونات، ثم يحدّثها لقيمة غير صفرية. يمكن للحدس اقتراح أن أسوأ نتيجة ممكنة لحالة التعارض أعلاه هي استدعاء وظيفة loadIcon عدّة مرات. وبينما أن الـ روتين-جو الأول مشغول بتحميل الأيقونات، فإن الـ روتين-جو الآخر الذين سيدخل Icon سيجد أن المتغير لا زال يساوي صفر، وسيستدعي أيضًا loadIcons.

هذا التخمين خاطئ أيضًا (نحن نأمل أن تكون طورّت حدسا جديدا بشأن التزامن الآن، وهو أن التخمينات بشأن التزامن لا يمكن الثقة فيها). تذكر مناقشة الذاكرة من القسم 9.4. في غياب المزامنة الواضحة، سيكون المترجم و CPU أحرار في إعادة طلب الدخول للذاكرة بأي عدد من الطرق، طالما أن سلوك كل روتين-جو متسق تتابعيًا. أحد احتمالات إعادة الطلب لعبارات loadIcon موضحة أدناه. وهي تخزّن خريطة فارغة في متغير الأيقونات قبل أن تملئه:

```
func loadIcons() {
	icons = make(map[string]image.Image)
	icons["spades.png"] = loadIcon("spades.png")
	icons["hearts.png"] = loadIcon("hearts.png")
```

```
    icons["diamonds.png"] = loadIcon("diamonds.png")
    icons["clubs.png"] = loadIcon("clubs.png")
}
```

من ثم، فإن روتين-جو الذي يجد أن الأيقونات غير صفرية قد لا يفترض أن تهيئة المتغير اكتملت.

أبسط طريقة صحية لضمان انتباه كل الـ روتينات جو لآثار loadIcons هو مزامنتها باستخدام mutex:

```
var mu sync.Mutex // guards icons
var icons map[string]image.Image

// Concurrency-safe.
func Icon(name string) image.Image {
    mu.Lock()
    defer mu.Unlock()
    if icons == nil {
        loadIcons()
    }
    return icons[name]
}
```

مع ذلك، فإن تكلفة إجبار الدخول الاستثنائي المتبادل على الأيقونات هو أنه لا يمكن لاثنين من روتين-جو الدخول للمتغير بالتزامن، حتى بعد تهيئة المتغير بأمان وعدم تعديله أبدًا مرة أخرى. يقترح هذا قفل القراء المتعددين:

```
var mu sync.RWMutex // guards icons
var icons map[string]image.Image

// Concurrency-safe.
func Icon(name string) image.Image {
    mu.RLock()
    if icons != nil {
        icon := icons[name]
        mu.RUnlock()
        return icon
    }
    mu.RUnlock()
    // acquire an exclusive lock
    mu.Lock()
    if icons == nil { // NOTE: must recheck for nil
        loadIcons()
    }
    icon := icons[name]
    mu.Unlock()
    return icon
}
```

هناك قسمان حَرِجان الآن. يحصل روتين-جو أولاً على قفل القارئ، ويستشير الخريطة، ثم يحرر القفل. إذا وجد أي مدخل (الحالة الشائعة)، فإنه يعود. وإذا لم يجد أي مدخلة، فإن روتين-جو يحصل على قفل كاتب. ولا يوجد طريقة لترقية قفل مشترك إلى قفل استثنائي بدون تحرير القفل المشترك أولا، وبالتالي يجب أن نعيد فحص متغير الأيقونات لنتأكد من عدم وجود روتين-جو آخر يهيئه خلال هذا الوقت.

يمنحنا النمط أعلاه تزامن أكبر، ولكنه معقد وبالتالي ميال للتعرض للأخطاء أكثر. لحسن الحظ، تقدم حزمة sync حل متخصص لمشكلة التهيئة لمرة واحدة: sync.Once. ونظريا، فإن Once يتكوّن من mutex ومتغير منطقي (Boolean) يسجل ما إذا كانت التهيئة حدثت أم لا، ويحمي الـ mutex كل من المتغير المنطقي وبنيات بيانات العميل. تقبل الطريقة المنفردة Do وظيفة التهيئة كمعطى لها. لنستخدم Once لتبسيط وظيفة Icon:

```
var loadIconsOnce sync.Once
var icons map[string]image.Image

// Concurrency-safe.
func Icon(name string) image.Image {
     loadlconsOnce.Do(loadlcons)
     return icons[name]
}
```

إن كل استدعاء لـ Do(loadIcons) يقفل mutex، ويفحص المتغير المنطقي. وفي الاستدعاء الأول، الذي كان المتغير خاطئًا فيه، تستدعي Do الـ loadIcons، وتضبط المتغير على أنه صحيح (true). لا تفعل الاستدعاءات اللاحقة أي شيء، ولكن مزامنة mutex تضمن أن آثار loadIcons على الذاكرة (وخاصة الأيقونات) ستصبح واضحة لكل روتينات جو. إن استخدام sync.Once بهذه الطريقة يمكّننا من تجنب مشاركة المتغيرات مع روتينات جو الأخرى حتى يتم بناءها بشكل مناسب.

تمرين 9.2: أعِد كتابة مثال PopCount من القسم 2.6.2 حتى يهيئ جدول الفحص باستخدام sync.Once في أول مرة يحتاجه. (من الناحية الواقعية، إن تكلفة المزامنة ستكون محظورة لوظيفة صغيرة وذات أداء مرتفع مثل PopCount).

9.6 كاشِف التعارض

إن ارتكاب أخطاء التزامن سهل جدًا حتى مع الانتباه الشديد لها. لحسن الحظ، فإن زمن تشغيل وسلسلة أدوات لغة Go مجهزين بأداة تحليل ديناميكية دقيقة وسهلة الاستخدام، وهي كاشِف التعارُض (Race Detector).

قم بإضافة عَلَم التعارض race- وحسب إلى أمر go build أو goo run أو go test. سيجعل هذا المترجم يبني نسخة معدلة من تطبيقك أو اختبارك مع أدوات إضافية تسجل بفعالية كل حالات الدخول للمتغيرات المشتركة التي حدثت

أثناء التنفيذ، إضافة إلى هوية روتين-جو الذي قرأ أو كتب المتغير. إضافة إلى ذلك، يسجّل البرنامج المعدّل كل أحداث المزامنة، مثل عبارات go، وعمليات القناة، واستدعاءات Lock.(sync.Mutex*) و Wait.(sync.WaitGroup*) وغيرهم. (إن مجموعة أحداث المزامنة الكاملة مُحددة في مستند " The Go Memory Model " المصاحب لمواصفات اللغة).

يدرس كاشف التعارض تسلسل الأحداث هذا، ويبحث عن الحالات التي يقرأ أو يكتب روتين-جو فيها متغيرا مشتركا كُتب مؤخرًا بواسطة روتين-جو مختلف بدون عملية مزامنة متداخلة. يشير هذا إلى دخول متزامن للمتغير المشترك، وبالتالي إلى وجود تعارض بيانات. تطبع الأداة تقرير يتضمن هوية المتغير، وكومة من استدعاءات الوظيفة النشطة في روتين-جو القراءة، و روتين-جو الكتابة. يكفي هذا عادة لتحديد المشكلة. يحتوي القسم 9.7 على مثال على عمل كاشف التعارض.

يقدّم كاشف التعارض تقرير بكل حالات تعارض البيانات التي نُفِذت فعليًا، ولكنه يستطيع كشف حالات التعارف فقط لو حدثت خلال التشغيل، ولا يمكنه إثبات أنه لن تحدث حالة تعارض أبدًا. ولتحقيق أفضل النتائج، تأكد من أن اختباراتك تنفيذ حِزمَك باستخدام التزامن.

يحتاج البرنامج المُدمج فيه كاشف تعارض إلى وقت وذاكرة أكثر للتشغيل بسبب حفظ السجلات الإضافي، ولكن النفقات الإضافية يمكن تحملها حتى بالنسبة للعديد من وظائف الإنتاج، حيث أن ترك كاشف التعارض يقوم بوظيفته يمكن أن يوفر العديد من الساعات والأيام في الكشف عن الأخطاء في الوظائف التي تحدث فيها حالات تعارض بشكل غير متكرر.

9.7 مثال: المخبأ المتزامن غير المانِع (Concurrent Non-Blocking Cache).

سنبني في هذا القسم "مخبأ متزامن غير مانع"، وهو تجريد يحل مشكلة تنشأ عادة في البرامج الحقيقية المتزامنة، ولكن لا تتناوله المكتبات الحالية جيدًا. إن هذه مشكلة وظيفة "التحسين" (memoizing)، بمعنى، تخبئة نتيجة وظيفة بحيث لا تحتاج للحساب إلا مرة واحدة. سيكون حلّنا آمنا تزامنيًا، وسيتجنب التنازع المرتبط بالتصميمات القائمة على قفل واحد للمخبأ بأكمله.

سنستخدم وظيفة httpGetBody أدناه كمثال على نوع الوظيفة التي قد نرغب في إجراء memorize لها. وهي تقدم طلب HTTP GET، وتقرأ جسم الطلب. إن استدعاء هذه الوظيفة مكلف نسبيًا، لذا سنرغب في تجنب تكرار الاستدعاء دون ضرورة.

```
func httpGetBody(url string) (interface{}, error) {
	resp, err := http.Get(url)
	if err != nil {
		return nil, err
	}
	defer resp.Body.Close()
	return ioutil.ReadAll(resp.Body)
}
```

يخفي السطر الأخير غموضا طفيفا. يعيد ReadAll نتيجتين، هما byte [] و error، ولكن حيث أن هذه النتائج يمكن نسبتها لأنواع النتيجة المعلَنة {}httpGetBody—interface و error، على الترتيب، بالتالي يمكننا أن نعيد نتيجة الاستدعاء دون مزيد من الضوضاء. لقد اخترنا نوع الإعادة هذا لـ httpGetBody حتى يلتزم بنوع الوظائف التي ضُمم مخبأنا لتحسينها (memorize).

إليك أول مسودة للمخبأ:

```
gopl.io/ch9/memo1
// Package memo provides a concurrency-unsafe
// memoization of a function of type Func.
package memo
// A Memo caches the results of calling a Func.
type Memo struct {
    f     Func
    cache map[string]result
}
// Func is the type of the function to memoize.
type Func func(key string) (interface{}, error)
type result struct {
    value interface{}
    err   error
}
func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]result)}
}
// NOTE: not concurrency-safe!
func (memo *Memo) Get(key string) (interface{}, error) {
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    return res.value, res.err
}
```

يثبِّت مثال Memo وظيفة f في memoize، من النوع Func، والمخبأ، وهو ترسيم خريطة من السلاسل إلى النتائج. إن كل نتيجة هي ببساطة زوج من النتائج التي يعيدها استدعاء f – وهي قيمة وخطأ. سنعرض تنويعات عديدة على Memo مع تقدُّم التصميم، ولكننا سنشارك كل هذه الجوانب الأساسية.

ستجد أدناه مثال على استخدام Memo، وسنستدعي Get لكل عنصر في تيار URLs الوارد، ونسجل زمن وصول الاستدعاء ومقدار البيانات التي يعيدها.

```
m := memo.New(httpGetBody)
for url := range incomingURLs() {
	start := time.Now()
	value, err := m.Get(url)
	if err != nil {
		log.Print(err)
	}
	fmt.Printf("%s, %s, %d bytes\n",
		url, time.Since(start), len(value.([]byte)))
}
```

يمكننا اختبار حزمة testing (موضوع الفصل 11) لبحث تأثير الـ memorization منهجيًا. وسنرى من ناتج الاختبار أدناه أن تيار URL يحتوي على مكررات، وأنه بالرغم من أن الاستدعاء الأول لـ Get.(Memo*) لكل URL يستغرق مئات المللي ثانية، إلا أن الطلب الثاني يعيد نفس مقدار البيانات خلال أقل من مللي ثانية.

```
$ go test -v gopl.io/ch9/memo1
=== RUN Test
https://golang.org, 175.026418ms, 7537 bytes
https://godoc.org, 172.686825ms, 6878 bytes
https://play.golang.org, 115.762377ms, 5767 bytes
http://gopl.io, 749.887242ms, 2856 bytes
https://golang.org, 721ns, 7537 bytes
https://godoc.org, 152ns, 6878 bytes
https://play.golang.org, 205ns, 5767 bytes
http://gopl.io, 326ns, 2856 bytes
--- PASS: Test (1.21s)
PASS
ok gopl.io/ch9/memo1 1.257s
```

ينفذ هذا الاختبار كل استدعاءات Get بالتتابع.

ونظرًا لكون طلبات HTTP هي فرصة رائعة للموازاة، لنغير الاختبار بحيث يقوم بكل الطلبات بالتزامن. يستخدم الاختبار sync.WaitGroup للانتظار حتى اكتمال الطلب الأخير قبل إعادته:

```
m := memo.New(httpGetBody)
var n sync.WaitGroup
for url := range incomingURLs() {
	n.Add(1)
	go func(url string) {
		start := time.Now()
		value, err := m.Get(url)
		if err != nil {
```

```
                log.Print(err)
            }
        fmt.Printf("%s, %s, %d bytes\n",
            url, time.Since(start), len(value.([]byte)))
        n.Done()
        }(url)
}
n.Wait()
```

يعمل الاختبار أسرع بكثير، ولكن لسوء الحظ من غير المرجح أن يعمل بطريقة صحيحة طوال الوقت. قد نلاحظ أخطاء مخبأ غير متوقعة، أو أن مخبأ يعيد قيما غير صحيحة، أو تنهار حتى.

الأسوأ من ذلك أنها من الممكن أن تعمل بطريقة صحيحة بعض الوقت، وبالتالي قد لا نلاحظ حتى أن بها مشكلة. ولكن لو شغلناها مع عَلَم race-، فإن كاشف التعارض (انظر 9.6) يطبع عادة تقرير كالتالي:

```
$ go test -run=TestConcurrent -race -v gopl.io/ch9/memo1
=== RUN TestConcurrent
WARNING: DATA RACE
Write by goroutine 36:
  runtime.mapassign1()
    ~/go/src/runtime/hashmap.go:411 +0x0
  gopl.io/ch9/memo1.(*Memo).Get()
    ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
  ...
Previous write by goroutine 35:
  runtime.mapassign1()
    ~/go/src/runtime/hashmap.go:411 +0x0
  gopl.io/ch9/memo1.(*Memo).Get()
    ~/gobook2/src/gopl.io/ch9/memo1/memo.go:32 +0x205
...
Found 1 data race(s)
FAIL     gopl.io/ch9/memo1 2.393s
```

إن الإشارة إلى memo.go:32 تخبرنا أن هناك اثنان من روتينات جو حدّثا خريطة المخبأ بدون أي تزامن متداخل. و Get ليست آمنة تزامنيًا، بل بها تعارض بيانات.

```
28  func (memo *Memo) Get(key string) (interface{}, error) {
29              res, ok := memo.cache[key]
30              if !ok {
31                      res.value, res.err = memo.f(key)
32                      memo.cache[key] = res
33              }
34          return res.value, res.err
35  }
```

إن أبسط طريقة لجعل المخبأ آمن تزامنيًا هي استخدام التزامن القائم على المراقبة. كل ما نحتاج لفعله هو إضافة mutex إلى Memo، والحصول على قفل mutex في بداية Get، وتحريره قبل عودة Get، بحيث تحدث عمليتيّ المخبأ داخل القسم الحَرج:

```
gopl.io/ch9/memo2
type Memo struct {
    f     Func
    mu    sync.Mutex // guards cache
    cache map[string]result
}
// Get is concurrency-safe.
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    if !ok {
        res.value, res.err = memo.f(key)
        memo.cache[key] = res
    }
    memo.mu.Unlock()
    return res.value, res.err
}
```

أصبح كاشف التعارض صامتا الآن، حتى عند إجراء الاختبارات بالتزامن. لسوء الحظ، يعكس هذا التغيير على Memo المكاسب التي حققناها من أداءنا السابق. وعند قيامنا بالاحتفاظ بالقفل طوال مدة كل استدعاء لـ f، تقوم Get بعمل تسلسل لكل عمليات I/O التي نهدف إلى موازاتها. إن ما نحتاجه هو مخبأ غير مانع، مخبأ لا يقوم بعمل تسلسل للاستدعاءات للوظيفة التي يحسّنها (memoizes).

سنرى في التطبيق التالي لـ Get الموضح أدناه أن استدعاء روتين-جو يؤدي للحصول على القفل مرتين: واحدة للبحث، ثم واحدة أخرى للتحديث لو لم يُعد البحث أي نتائج. وفيما بين الاثنين، ستكون روتينات جو الأخرى حرة في استخدام المخبأ.

```
gopl.io/ch9/memo3

func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    res, ok := memo.cache[key]
    memo.mu.Unlock()
    if !ok {
        res.value, res.err = memo.f(key)
        // Between the two critical sections, several goroutines
        // may race to compute f(key) and update the map.
        memo.mu.Lock()
        memo.cache[key] = res
        memo.mu.Unlock()
    }
    return res.value, res.err
}
```

يتحسّن الأداء مرة أخرى، ولكننا نلاحظ الآن أن بعض الـ URLs يتم جلبها مرتين. يحدث هذا عندما تستدعي روتينيَ جو أو أكثر Get لنفس الـ URL في نفس الوقت تقريبًا. يستشير كلاهما المخبأ، ولا يجدان قيمة هناك، ثم يستدعيان وظيفة f البطيئة. ثم يقوم كل منهما بتحديث الخريطة بالنتيجة التي حصلوا عليها. أحد النتائج تكتب فوقها نتيجة أخرى.

سنرغب عادة في تجنب هذا العمل الزائد، وهذه الخاصية يُطلق عليها أحيانًا "الكبت المتكرر" (duplicate suppression). سنرى في نسخة Memo أدناه أن كل عنصر في الخريطة هو مؤشر لـ entry struct، ويحتوي كل entry على النتيجة المُحسّنة (memorized) الناتجة عن استدعاء الوظيفة f، كما حدث من قبل، ولكنها تحتوي إضافة إلى ذلك على قناة اسمها ready. بمجرد تحديد نتيجة entry، ستُغلق هذه القناة، لتبث (انظر 8.9) لأي روتينات جو أخرى أنه من الآمن الآن أن يقرأوا النتيجة من entry.

```
gopl.io/ch9/memo4

type entry struct {
    res   result
    ready chan struct{} // closed when res is ready
}
func New(f Func) *Memo {
    return &Memo{f: f, cache: make(map[string]*entry)}
}
type Memo struct {
    f     Func
    mu    sync.Mutex // guards cache
    cache map[string]*entry
}
func (memo *Memo) Get(key string) (value interface{}, err error) {
    memo.mu.Lock()
    e := memo.cache[key]
    if e == nil {
        // This is the first request for this key.
        // This goroutine becomes responsible for computing
        // the value and broadcasting the ready condition.
        e = &entry{ready: make(chan struct{})}
        memo.cache[key] = e
        memo.mu.Unlock()
        e.res.value, e.res.err = memo.f(key)
        close(e.ready) // broadcast ready condition
    } else {
        // This is a repeat request for this key.
        memo.mu.Unlock()
        <-e.ready // wait for ready condition
    }
    return e.res.value, e.res.err
}
```

يتضمن استدعاء Get الآن الحصول على قفل mutex الذي يحمي خريطة المخبأ، والبحث داخل الخريطة عن مؤشر لـ entry موجود، وتخصيص وإدخال entry جديد لو لم يتم إيجاد واحد، ثم تحرير القفل. لو كان هناك entry موجود بالفعل، فقد لا تكون قيمته جاهزة بالضرورة بعد – قد يكون هناك روتين-جو لا زال يستدعي وظيفة f البطيئة – لذا فإن

روتين-جو المُستدعي يجب أن ينتظر حالة "استعداد" الـ entry قبل أن يقرأ نتيجة الـ entry. يفعل هذا من خلال قراءة القيمة من قناة الاستعداد، حيث أن العملية تُحظَر حتى إغلاق القناة.

لو لم يكن هناك entry موجود، فإن إدخال entry "غير مستعد" في الخريطة سيجعل الـ روتين-جو الحالي مسئول عن استثارة الوظيفة البطيئة، وتحديث الـ entry، وبث استعداده لـ entry جديد لأي روتين-جو آخر قد يكون مُنتظِرًا حينها.

لاحظ أن المتغيرات e.res.value و e.res.err في entry مشتركين بين روتينات جو متنوعة. إن روتين-جو الذي يصنع الـ entry يحدد قيمها، وتقرأ روتينات جو الأخرى قيمها بمجرد بث حالة "الاستعداد". وبالرغم من دخولها بواسطة روتينات جو متعددة، إلا أننا لن نحتاج لقفل mutex بالضرورة. ويحدث إغلاق قناة الاستعداد قبل استقبال أي روتين-جو آخر لحدث البث، لذا فإن الكتابة على هذه المتغيرات في روتين-جو الأول تحدث قبل قراءتها بواسطة روتينات جو التالية. وبالتالي لا يوجد تعارض بيانات.

اكتمل مخبأنا المتزامن، غير المانع، ذو الكبت المتكرر الآن.

إن تطبيق Memo أعلاه يستخدم mutex لحراسة متغير الخريطة الذي يشاركه كل روتين-جو يستدعي Get. ومن المثير للاهتمام مقارنة هذا التصميم مع التصميم البديل الذي يُقيّد فيه متغيّر الخريطة مع روتين-جو مراقبة يجب أن يرسل له مستدعين Get رسالة.

ستظل إعلانات Func و result و entry كما كانت من قبل:

```
// Func is the type of the function to memoize.
type Func func(key string) (interface{}, error)

// A result is the result of calling a Func.
type result struct {
	value interface{}
	err error
}
type entry struct {
	res result
	ready chan struct{} // closed when res is ready
}
```

مع ذلك، يتكوّن نوع Memo الآن من قناة، وطلبات، يتواصل مُستدعي Get من خلالها مع روتين-جو المراقبة. إن نوع عنصر القناة هو طلب. وباستخدام هذه البِنية، يرسل مستدعي Get إلى روتين-جو المراقبة كلا المفتاحين، بمعنى أنه يرسل مُعطى لوظيفة memoizes، وقناة أخرى واستجابة أخرى تُرسل من خلالها النتيجة مرة أخرى عند توافرها. ستحمل هذه القناة قيمة واحدة فقط.

```
gopl.io/ch9/memo5
```

```
// A request is a message requesting that the Func be applied to key.
type request struct {
    key      string
    response chan<- result // the client wants a single result
}
type Memo struct{ requests chan request }
// New returns a memoization of f.  Clients must subsequently call Close.
func New(f Func) *Memo {
    memo := &Memo{requests: make(chan request)}
    go memo.server(f)
    return memo
}
func (memo *Memo) Get(key string) (interface{}, error) {
    response := make(chan result)
    memo.requests <- request{key, response}
    res := <-response
    return res.value, res.err
}
func (memo *Memo) Close() { close(memo.requests) }
```

تصنع طريقة Get أعلاه قناة استجابة، وتضعها في الطلب، وترسلها إلى روتين-جو المراقبة، ثم تستلمها منه فورًا.

إن متغير المخبأ مقيد بـ روتين-جو المراقبة (*Memo).server الموضّح أدناه. يقرأ المراقب الطلبات في حلقة حتى تُغلق القناة الطالبة بواسطة طريقة Close. وهي ترجع للمخبأ مع كل طلب، وتُنشئ وتُدخِل entry جديد لو لم يتم إيجاد واحد.

```
func (memo *Memo) server(f Func) {
    cache := make(map[string]*entry)
    for req := range memo.requests {
        e := cache[req.key]
        if e == nil {
            // This is the first request for this key.
            e = &entry{ready: make(chan struct{})}
            cache[req.key] = e
            go e.call(f, req.key) // call f(key)
        }
        go e.deliver(req.response)
    }
}
func (e *entry) call(f Func, key string) {
    // Evaluate the function.
    e.res.value, e.res.err = f(key)
    // Broadcast the ready condition.
    close(e.ready)
}
func (e *entry) deliver(response chan<- result) {
    // Wait for the ready condition.
    <-e.ready
    // Send the result to the client.
    response <- e.res
}
```

كما هو الحال في النسخة المعتمدة على mutex، فإن الطلب الأول لمفتاح معين يصبح مسئول عن استدعاء الوظيفة f في هذا المفتاح، وتخزين النتيجة في entry، وبث استعداد الـ entry من خلال إغلاق قناة ready. يتم هذا من خلال (*entry).call

يجد الطلب التالي لنفس المفتاح entry موجود في الخريطة وينتظر النتيجة كي يصبح مستعدًا، ويرسل النتيجة عبر قناة response إلى روتين-جو العميل الذي استدعى Get. يتم هذا من خلال (*entry).deliver. يجب استدعاء طرق call و deliver من خلال روتينات جو الخاصة بهم لضمان أن روتين-جو المراقبة لا يوقف معالجة الطلبات الجديدة.

يوضح هذا المثال أنه يمكن بناء العديد من البِنيات المتزامنة باستخدام طريقة من الاثنتين – المتغيرات المشتركة والأقفال، أو توصيل العمليات التتابعية – دون تعقيد زائد عن الحد.

ليس من الواضح دائمًا ما هي الطريقة المفضلة في موقف معين، ولكن ملاحظة طريقة استجابتهم هو أمر جدير بالاهتمام. وقد يؤدي التنقل من طريقة إلى أخرى إلى تبسيط الشفرة.

تمرين 9.3: قم بتوسيع نوع func وطريقة (*Memo).Get حتى يتمكّن المستدعين من توفير قناة done اختيارية يمكنهم إلغاء العملية عن طريقها (انظر 8.9). لا يجب تخبأة نتائج استدعاء Func المُلغى.

9.8 روتينات جو والخيوط (Threads)

ذكرنا في الفصل السابق أن الفارق بين روتينات جو وخيوط نظام التشغيل (OS) يمكن تجاهله حتى وقت لاحق. وبالرغم من أن الفوارق بينهما كمية في الأساس، إلا أن الفارق الكمي الكبير بما يكفي يمكن أن يصبح فارقا نوعيا، كذلك الحال أيضًا مع روتينات جو والخيوط. و الآن قد حان الوقت للتفريق بينهما.

9.8.1 الرصّات القابلة للنمو (Growable Stacks).

يحتوي كل خيط OS على كتلة ذاكرة ثابتة الحجم (عادة ما تصل إلى 2 ميجا بايت) في "رصّته" (stack)، وهي منطقة العمل التي يحفظ فيها المتغيرات المحلية لاستدعاءات الوظيفة التي لا زالت قيد العمل أو الموقوفة مؤقتًا بينما تستدعى وظيفة أخرى. إن هذه الرصّة ثابتة الحجم أكبر من اللازم وأقل من اللازم في وقت واحد. إن الرصّة التي يصل حجمها لـ 2 ميجا بايت ستكون بمثابة هدر هائل للذاكرة على روتين-جو صغير، كالذي ينتظر WaitGroup وحسب ثم يغلق القناة. ومن الشائع أن يصنع برنامج Go مئات الآلاف من روتينات جو في وقت واحد، وهو أمر سيصبح مستحيل

مع رصّات بهذا الحجم. مع ذلك، فإن الرصّات ثابتة الحجم قد لا تكون كبيرة بما يكفي دائمًا لتناسب الوظائف المعقدة جدًا وشديدة التكرار. يمكن لتغيير الحجم الثابت تحسين كفاءة المساحة، والسماح بإنشاء المزيد من التشعبات، أو يمكن أن يسمح بالمزيد من الوظائف التكرارية الثقيلة، ولكن لا يمكنه فعل كلاهما في وقت واحد.

على النقيض، يبدأ روتين-جو حياته برصّة صغيرة، عادة ما يكون حجمها 2 كيلو بايت، ورصّة روتين-جو، مثلها مثل تشعب OS، تحمل متغيرات محلية خاصة باستدعاءات الوظائف النشطة والموقوفة، ولكن على العكس من تشعب OS، رصّة الـ روتين-جو غير ثابتة، وهي تنمو وتنكمش حسب الحاجة. إن حَد حجم رصة روتين-جو يمكن أن يصل إلى 1 جيجا بايت، وتكون خاصة بأوامر ذات أحجام أكبر من رصة التشعب ذات الحجم الثابت التقليدية، ولكن القليل فقط من روتينات جو تستخدم كل هذا الحجم بالطبع.

تمرين 9.4: قم بإنشاء توارد يربط عدد اعتباطي من روتينات جو مع القنوات. ما هو أقصى عدد من مراحل التوارد يمكنك إنشاءها بدون نفاذ الذاكرة؟ كَم تستغرق القيمة لنقل التوارد بأكمله؟

9.8.2 جَدولة روتين-جو

تُجدوَل خيوط النظام بواسطة نواة النظام. ويقوم مؤقّت عتاد بمقاطعة المعالج كل بضعة مللي ثوان، وهو ما يسمح لوظيفة نواة اسمها scheduler أن تستدعى. توقف هذه الوظيفة الخيط الذي يقوم بالتنفيذ حاليًا، وتحفظ سجلاته في الذاكرة، وتفحص قائمة الخيوط، وتقرر أيها يجب أن يعمل تاليًا، وتستعيد سجلات هذا الخيط من الذاكرة، ثم تتابع تنفيذ الخيط. ونظرًا لكون خيوط النظام تُجدوَل بواسطة النواة، فإن تمرير التحكم من خيط إلى آخر سيتطلب "تحويل سياق" (context switch) كامل، أي حفظ حالة خيط مُستخدم واحد في الذاكرة، واسترجاع حالة خيط مستخدم آخر، وتحديث بِنيات بيانات المُجدوِل. إن هذه العملية بطيئة بسبب محليتها الضعيفة، وعدد مُدخلات الذاكرة المطلوبة، وقد ساءت أكثر تاريخيًا مع زيادة عدد دورات CPU لتي تطلب الدخول للذاكرة.

يحتوي زمن تشغيل Go على مُجدوله الخاص الذي يستخدم تقنية تُعرف بـ m:n scheduling، لأنه يضاعِف (أو يُجدول) روتينات جو (m) في خيوط (n). إن وظيفة مُجدول جو مناظرة لوظيفة مُجدول النواة، ولكنها تهتم فقط بـ روتينات جو لبرنامج Go منفرد.

لا يستدعى مُجدول Go دوريًا بواسطة مؤقّت العتاد – وهذا على النقيض من مُجدول خيوط نظام التشغيل – بل يُحفز ضمنيًا بواسطة بنيات لغة Go معينة. كمثال، عندما يستدعي روتين-جو time.Sleep أو مكمّلات في قناة أو عملية mutex، فإن المُجدول يضعه في وضع النوم، ويشغّل روتين-جو آخر حتى يحين وقت إيقاظ روتين-جو الأول. إن إعادة جدولة روتين-جو أرخص كثيرًا من إعادة جدولة الخيط لأنه لا يحتاج للانتقال إلى سياق النواة.

تمرين 9.5: اكتب برنامجا فيه اثنان من روتينات جو يرسل رسائل ما بين قناتين غير صوانيتين بطريقة البينج-بونج. كم عدد الاتصالات في الثانية التي يمكن للبرنامج الحفاظ عليها؟

9.8.3 GOMAXPROCS

يستخدم مُجدوِل Go معامل اسمه GOMAXPROCS لتحديد عدد خيوط النظام التي يمكنها تنفيذ شفرة Go بالتزامن وبشكل فعال. إن قيمته الاعتيادية هي عدد CPUs في الآلة، وبالتالي ففي الآلة التي تحتوي على 8 CPUs، سيُجدول المُجدول شفرة Go على ما يصل إلى 8 خيوط OS في وقت واحد. (إن GOMAXPROCS هو الـ n في جدولة m:n). إن روتينات جو النائمة أو المحظورة في الاتصال لا تحتاج لتشعب على الإطلاق. وروتينات جو المحظورة في I/O أو استدعاءات أي نظام آخر أو التي تستدعي وظائف بخلاف go، تحتاج إلى خيط نظام، ولكن GOMAXPROCS غير مسئول عنهم.

يمكنك التحكم في هذا المعامل صراحةً باستخدام متغير بيئة GOMAXPROCS أو وظيفة runtime.GOMAXPROCS. ويمكننا رؤية تأثير GOMAXPROCS على هذا البرنامج الصغير الذي يطبع تيار لا ينتهي من الأصفار والواحد:

```
for {
    go fmt.Print(0)
    fmt.Print(1)
}
```

```
$ GOMAXPROCS=1 go run hacker-cliche.go
 111111111111111111110000000000000000000011111...
$ GOMAXPROCS=2 go run hacker-cliche.go
 010101010101010101011001100101011010010100110...
```

يُنفذ روتين-جو واحد فقط على الأكثر في أول تشغيل، ويكون روتين-جو الرئيسي في البداية، ويقوم بطباعة قيم الواحد. وبعد فترة من الوقت، يُدخله مُجدول Go في حالة نوم ويوقظ روتين-جو يطبع الأصفار، ويمنحه دورًا في العمل على خيط النظام. وفي الدور الثاني، كان هناك خيطين متاحين، وبالتالي عمل كلا روتينات جو بالتزامن، وطَبعا الأرقام بنفس المعدل تقريبًا. يجب أن نؤكد على أن هناك العديد من العوامل التي تدخل في جدولة روتين-جو، وأن زمن التشغيل يتطور باستمرار، وبالتالي قد تختلف نتائجك عن النتائج المُقدّمة أعلاه.

تمرين 9.6: قِس كيف يتفاوت أداء برنامج متوازي مقيّد بالحساب (انظر التمرين 8.5) مع استخدام GOMAXPROCS. وما هي القيمة المثالية على كمبيوترك؟ وكم عدد الـ CPUs التي يحتوي عليها الكمبيوتر الخاص بك؟

9.8.4 روتينات جو ليس لها هوية

إن معظم نظم التشغيل ولغات البرمجة التي تدعم الخيوط المتعددة، يمتلك فيها الخيط الحالي هوية مميزة يمكن الحصول عليها بسهولة كقيمة عادية، والتي عادة ما تكون عدد صحيح أو مؤشّر. يجعل هذا من السهل بناء تجريد اسمه "التخزين المحلي للخيط" (thread-local storage)، وهو خريطة عالمية مفاتيحها هي هوية الخيط، بحيث يتمكّن كل خيط من تحزين واستعادة القيم بشكل مستقل عن الخيوط الأخرى.

لا تمتلك روتينات جو مفهوم هوية يمكن للمبرمِج الوصول إليها. وقد فُعِل هذا عمدًا في التصميم، حيث أن التخزين المحلي للخيوط يُساء استخدامه عادة. على سبيل المثال، من الشائع جدًا في خادم الويب المُطبّق بلغة تحتوي على تخزين محلي للخيط أن تجد العديد من الوظائف معلومات حول طلب HTTP الذي تعمل نيابة عنه حاليًا من خلال البحث في المخزون. مع ذلك، وكما هو الحال في البرامج التي تعتمد بإفراط على المتغيرات العالمية، يمكن أن يؤدي هذا إلى "عمل عن بُعد" غير صحي، لا يتحدد فيه سلوك الوظيفة بالمعطيات وحدها، وإنما من خلال هوية الخيط الذي تعمل عليه. ونتيجة لهذا، لو تغيرت هوية الخيط فإن الوظيفة قد يسوء أداؤها بشكل غير مفهوم.

تشجع لغة Go على أسلوب البرمجة الأبسط، والذي تكون فيه العوامل التي تؤثر على سلوك الوظيفة عوامل واضحة. يجعل هذا البرنامج سهل القراءة، ويجعلنا نحدد المهام الفرعية الخاصة بوظيفة معينة إلى روتينات جو عديدة مختلفة دون القلق بشأن هويتها.

لقد تعلمت الآن كل شيء حول خصائص اللغة التي تحتاجها لكتابة برامج Go، وسنعود في الفصلين التاليين لنلقي نظرة على الممارسات والأدوات التي تدعم البرمجة على نطاق أوسع: كيفية بناء مشروع كمجموعة من الحزم، وكيف نحصل ونبني ونختبر ونعاير ونفرز ونوثق ونشارك تلك الحِزَم.

10- الجِزَم وأداة Go

قد يحتوي برنامج متواضع الحجم اليوم على حوالي 10,000 وظيفة، ولكن صانع البرنامج لن يحتاج إلا إلى التفكير في عدد قليل منهم، وتصميم عدد أقل مما فكر فيه، لأن الأغلبية العظمى كتبها آخرون ومتاحة بسهولة عبر الحِزم (packages).

تأتي Go مع أكثر من 100 حزمة قياسية تُقدّم أُسس لمعظم التطبيقات. يُعد مجتمع Go بيئة مزدهرة بتصميم الحِزم والمشاركة وإعادة الاستخدام والتحسين، وقد نشر المجتمع الكثير جدًا من الأشياء، يمكنك إجراء بحث عليها في الفهرس الموجود على: http://godoc.org. سنوضح في هذا الفصل كيف يمكننا استخدام الحزم الموجودة وصُنع حزم جديدة.

تأتي لغة Go أيضًا مع أداة go، وهي أوامر دقيقة ولكنها سهلة الاستخدام لإدارة مساحات العمل في حزم Go. لقد أوضحنا منذ بداية هذا الكتاب كيف يمكننا استخدام أداة go لتحميل وبناء وتشغيل برامج مختلفة قدمناها كأمثلة. وسنلقي نظرة في هذا الفصل على المفاهيم الضمنية في الأداة، ونقوم بجولة نستعرض فيها إمكانياتها، والتي تتضمن طباعة المستندات، والبحث عن البيانات الوصفية حول الحزم في مساحة العمل. وسنستكشف في الفصل التالي خصائص الاختبار الموجودة فيها.

10.1 مُقدمة

إن هدف أي نظام حزمة هو جعل تصميم وصيانة البرامج الكبيرة أمرًا عمليًا من خلال تجميع الخصائص المرتبطة ببعضها معًا في وحدات يمكن فهمها وتغييرها بسهولة، وبشكل مستقل عن حزم البرنامج الأخرى. يسمح هذا التركيب بمشاركة الحزم وإعادة استخدامها بواسطة مشاريع مختلفة، موزّعة داخل المنظمة، أو متاحة للعالم بشكل عام.

تُعرِّف كل حزمة مساحة اسم مميزة تضم مُحدّداتها، ويرتبط كل اسم مع حزمة محددنا، مما يجعلنا نختار أسماء قصيرة وواضحة للأنواع والوظائف إلخ التي نستخدمها بشكل متكرر، بدون إحداث تضارب مع الأجزاء الأخرى للبرنامج.

تقدّم الحزم أيضًا "التغليف-encapsulation" من خلال التحكم في الأسماء الظاهرة أو المُصدّرة لخارج الحزمة. إن تقييد ظهور أعضاء الحزمة يخفي وظائف المُساعِد، والأنواع الموجودة وراء API الحزمة، مما يسمح لمُصين (maintainer) الحزمة بتغيير التطبيق واثقًا أن التغيير لن يؤثر على أي شفرة خارج الحزمة. إن تقييد الظهور يخفي المتغيرات أيضًا بحيث يتمكن العملاء من الدخول وتحديثهم فقط من خلال الوظائف المُصدّرة التي تحافظ على الثوابت الداخلية، أو تطبيق الاستثناء المتبادَل في البرنامج المتزامن.

عندما نغيّر ملفا، يجب أن نعيد تجميع حزمة الملف ومعها كل الحزم التي تعتمد عليها أيضًا. إن تجميع Go أسرع بشكل ملحوظ من معظم اللغات المُجمّعة الأخرى، حتى عند بناءه من الصفر. هناك ثلاث أسباب رئيسية لسرعة المُجمِّع. أولا، أن كل الواردات يجب أن تُدرج بشكل صريح في بداية كل ملف مصدر، وبالتالي لا يضطر المترجم لقراءة ومعالجة الملف بأكمله لتحديد اعتمادياته (dependencies). ثانيًا، تُشكّل اعتماديات الحزمة رسم بياني لا دوري (acyclic graph)، ونظرًا لعدم وجود دورات، يمكن تجميع الحزم بشكل منفصل وربما يمكن تجميعها بالتوازي كذلك.

أخيرًا، ملف الكائن الخاص بحزمة Go المُجمّعة يسجل معلومات التصدير، ليس للحزمة وحدها وحسب، ولكن لاعتمادياتها أيضًا. وعند تجميع حزمة، يجب أن يقرأ المُجمّع ملف كائن واحد لكل استيراد، ولكنه لا يحتاج للنظر فيما هو أبعد من هذه الملفات.

10.2 مسارات الاستيراد - Import Paths

تُحدد كل حزمة بواسطة سلسلة متفردة يُطلق عليها "مسار الاستيراد". ومسارات الاستيراد هي سلاسل تظهر في إعلانات الاستيراد.

```
import (
	"fmt"
	"math/rand"
	"encoding/json"
	"golang.org/x/net/html"
	"github.com/go-sql-driver/mysql"
)
```

كما ذكرنا في القسم 2.6.1، لا تحدد مواصفات لغة Go معنى هذه السلاسل، أو كيفية تحديد مسار استيراد الحزمة، ولكنها تترك هذه المسائل للأدوات. سنلقي نظرة مفصّلة في هذا الفصل على كيف تقوم أداة go بتفسيرهم، حيث أن أغلبية مبرمجي Go يستخدمونها لبناء واختبار برامجهم. هناك أدوات أخرى أيضًا

موجودة. كمثال، يستخدم مبرمجو Go نظام بناء جوجل الداخلي متعدد اللغات ويتبعون قواعد مختلفة في تسمية وتحديد موضع الحزم، وتحديد الاختبارات إلخ، لتناسب قواعد هذا النظام أكثر.

يجب أن تكون مسارات الاستيراد متفردة عالميًا في الحزم التي تنوي مشاركتها أو نشرها. ولتجنب التضاربات، يجب أن تبدأ مسارات استيراد كل الحزم المختلفة عن المكتبة الاعتيادية باسم نطاق الإنترنت الخاص بالمنظمة التي تملك أو تستضيف الحزمة، وهذا يجعل إيجاد الحزم ممكن كذلك. على سبيل المثال، يستورد الإعلان أعلاه مُحلل HTML بواسطة فريق Go، وتعريف قاعدة بيانات MySQL شعبي كطرف ثالث.

10.3 إعلان الحزمة

إن إعلان الحزمة مطلوب في بداية كل ملف مصدر في Go، وهدفه الأساسي هو تحديد المُعرِّف الاعتيادي لهذه الحزمة (يُطلق عليه "اسم الحزمة") عند استيراده بواسطة حزمة أخرى.

على سبيل المثال، كل ملف في حزمة math/rand يبدأ بالحزمة rand، وبالتالي حينما تستورد هذه الحزمة، يمكنك الدخول إلى أعضاءها كـ rand.Int,rand.Float64، وهكذا.

```
package main
import (
	"fmt"
	"math/rand"
)
func main() {
	fmt.Println(rand.Int())
}
```

إن اسم الحزمة عادة ما يكون آخر قطعة في مسار الاستيراد، ونتيجة لهذا، قد يكون لحزمتين نفس الاسم بالرغم من أن مسارات استيراداتهم مختلفة بالضرورة. كمثال، الحزم التي مسارات استيرادها math/rand و crypto/rand كلاهما اسمه rand. وسنرى كيف يمكننا استخدام كلاهما في نفس البرنامج في الجزء التالي.

هناك ثلاث استثناءات أساسية لاتفاق "القطعة الأخيرة/Last Segment". الاستثناء الأول هو أن الحزمة تُعرّف أمر (برنامج Go قابل للتنفيذ) ذو اسم يحتوي على الاسم الرئيسي دائمًا، بغض النظر عن مسار استيراد الحزمة. إن هذه إشارة على go build (انظر 10.7.3) الذي يبني مُوصِّل (Linker) لصنع ملف تنفيذي.

الاستثناء الثاني هو أن بعض الملفات في الدليل قد يُلحق بها كلمة _test في اسم الحزمة الخاصة بها لو كان اسم الملف ينتهي بـ test.go_. هذا الدليل يمكن أن يُعرّف حزمتين: الحزمة المعتادة، بالإضافة إلى حزمة أخرى اسمها "حزمة الاختبار الخارجي- external test package". إن لاحقة test_ ترسل رسالة إلى go test بأنه

يجب أن يبني كلتا الحزمتين، وهو تشير إلى أي الملفات ينتمي لكل حزمة. تُستخدم حزم الاختبار الخارجية لتجنب الدورات في رسم الاستيراد البياني الناتج عن اعتماديات الاختبار، وسنناقشها بالتفصيل أكثر في القسم 11.2.4.

الاستثناء الثالث هو أن بعض الأدوات الخاصة بإدارة الاعتمادية تضيف لواحق رقم الإصدار إلى مسارات استيراد الحزمة، مثل "gopkg.in/yaml.v2". يستثني اسم الحزمة اللاحقة، وفي هذه الحالة سيصبح فقط yaml.

10.4 إعلانات الاستيراد

إن ملف المصدر في لغة Go يحتوي على عدد صفر أو أكثر من إعلانات الاستيراد بعد إعلان الحزمة مباشرة، وقبل أول إعلام غير متعلق بالاستيراد. يمكن لكل إعلان استيراد أن يحدد مسار استيراد حزمة منفردة أو حزم متعددة في القائمة بين القوسين. إن الشكلين أدناه متكافئين ولكن الشكل الثاني أكثر شيوعا.

```
import "fmt"
import "os"

import (
     "fmt"
     "os"
)
```

يمكن تجميع الحزم المستوردة من خلال إدخال سطور فارغة، مثل المجموعات التي تشير لنطاقات مختلفة عادة. إن الترتيب غير مهم، ولكن حسب الاتفاق والسائد، فإن خطوط كل مجموعة تُرتّب أبجديًا. (كل من gofmt و goimports سيُجمعا ويُرتبا لأجلك).

```
import (
     "fmt"
     "html/template"
     "os"

     "golang.org/x/net/html"
     "golang.org/x/net/ipv4"
)
```

لو احتجنا لاستيراد حزمتين أسمائهما متماثلة، مثل math/rand و crypto/rand في حزمة ثالثة، فإن إعلان الاستيراد يجب أن يحدد اسم بديل لواحدة منهما على الأقل لتجنب التضارب. يُطلق على هذا "إعادة تسمية الاستيراد/renaming import".

```
import (
    "crypto/rand"
    mrand "math/rand" // alternative name mrand avoids conflict
)
```

يؤثر الاسم البديل على الملف المستوردِ فقط، بينما الملفات الأخرى..حتى الملفات الموجودة في نفس الحزمة، يمكن أن تستورد الحزمة باستخدام اسمها الاعتيادي أو اسم مختلف.

قد يكون "إعادة تسمية الاستيراد" مفيد حتى لو لم يكن هناك تضارب. لو كان اسم الحزمة المُستورَدة غير عملي، كما هو الحال أحيانًا في بعض الشفرات المكتوبة آليًا، فقد يكون الاسم المختصر ملائم أكثر. يجب استخدام نفس الاسم المختصر باستمرار لتجنب الارتباك. ويمكن لاختيار اسم بديل أن يساعد على تجنب التعارض مع أسماء المتغير المحلية الشائعة. كمثال، في الملف الذي يحتوي على العديد من المتغيرات المحلية التي تحتوي كلمة path في اسمها، يمكننا استيراد حزمة "path" قياسية كـ pathpkg.

يؤسس كل إعلان استيراد لاعتمادية من الحزمة الحالية إلى الحزمة المستورَدة. وتقدم أداة go build تقرير بالخطأ لو شكّلت هذه الاعتماديات دورة.

10.5 الاستيرادات الفارغة

من الخطأ استيراد حزمة إلى ملف دون الإشارة للاسم الذي تُعرّفه داخل هذا الملف. مع ذلك، يجب أن نستورد حزمة أحيانًا فقط من أجل الآثار الجانبية الناتجة عن فعل هذا: وهي تقييم تعبيرات المُهيِّئ (initializer) للمتغيرات على مستوى الحزمة، وتنفيذ وظائف init الخاصة بها (انظر 2.6.2). لكبت خطأ "الاستيراد غير المستخدم" الذي سنواجهه، يجب أن نستخدم خاصية إعادة تسمية الاستيراد والتي يكون الاسم البديل فيها هو _ أو المُعرِّف الفارغ (blank identifier). كالعادة، لا يمكن الإشارة للمُعرِّف الفارغ أبدًا.

```
import _ "image/png" // register PNG decoder
```

يُعرف هذا بالاستيراد الفارغ (blank import). وهو يُستخدم عادة لتطبيق آلية تجميع الوقت، حيث يمكن للبرنامج الرئيسي أن يتيح خصائص اختيارية من خلال الاستيراد الفارغ لحزم إضافية. أولا، سنرى كيفية استخدامه، ثم سنرى كيف يعمل.

إن حزمة Image القياسي في المكتبة تُصدّر وظيفة decode تقرأ البايتات من io.Reader، وتكتشف أي صيغة صورة استُخدمت لفك شفرة البيانات، وتستدعي أداة فك الشفرة المناسب، ثم تعيد image.Image الناتجة.

من السهل بناء مُحوّل صورة بسيطة يقرأ الصورة في صيغة ويكتبها في صيغة أخرى باستخدام .image.Decode

```go
gopl.io/ch10/jpeg
// The jpeg command reads a PNG image from the standard input
// and writes it as a JPEG image to the standard output.
package main
import (
	"fmt"
	"image"
	"image/jpeg"
	_ "image/png" // register PNG decoder
	"io"
	"os"
)
func main() {
	if err := toJPEG(os.Stdin, os.Stdout); err != nil {
		fmt.Fprintf(os.Stderr, "jpeg: %v\n", err)
		os.Exit(1)
	}
}
func toJPEG(in io.Reader, out io.Writer) error {
	img, kind, err := image.Decode(in)
	if err != nil {
		return err
	}
	fmt.Fprintln(os.Stderr, "Input format =", kind)
	return jpeg.Encode(out, img, &jpeg.Options{Quality: 95})
}
```

لو غذينا ناتج gopl.io/ch3/Mandelbrot (انظر 3.3) لبرنامج المُحوّل، فإنه سيكشف صيغة مُدخل PNG ويكتب نسخة JPEG من الشكل 3.3.

```
$ go build gopl.io/ch3/mandelbrot
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
Input format = png
```

لاحظ الاستيراد الفارغ لـ image/png، بدون هذا السطر سيجمع البرنامج ويربط كالعادة، ولكنه لن يتمكن من إدراك أو فك شفرة المُدخل الذي بصيغة PNG:

```
$ go build gopl.io/ch10/jpeg
$ ./mandelbrot | ./jpeg >mandelbrot.jpg
jpeg: image: unknown format
```

هكذا سيعمل الأمر. تقدم المكتبة القياسية أدوات لفك الشفرة لـ GIF و PNG و JPEG، ويمكن أن يقدم المستخدمين أدوات أخرى، ولكن لإبقاء الملفات التنفيذية صغيرة، يجب ألا تدرج أدوات فك الشفرة في

التطبيق ما لم تُطلب بشكل صريح. تستشير وظيفة image.Decode جدول من الصيغ المدعومة. ويحدد كل مُدخَل في الجدول أربع أشياء: اسم الصيغة، وسلسلة تعتبر سابقة (prefix) لكل الصور المُشفّرة بهذه الطريقة، والمستخدمة لكشف التشفير، ووظيفة Decode التي تفك شفرة صورة مشفرة، ووظيفة أخرى هي DecodeConfig التي تفك شفرة الصورة الموجودة في البيانات الوصفية فقط، مثل حجمها ولونها ومساحتها. يُضاف مُدخَل إلى الجدول من خلال استدعاء image.RegisterFormat، ويُستدعى عادة من داخل مهيّئ الحزمة الخاص بحزمة دعم كل صيغة، كتلك الموجودة في image/png:

```
package png // image/png
func Decode(r io.Reader) (image.Image, error)
func DecodeConfig(r io.Reader) (image.Config, error)
func init() {
     const pngHeader = "\x89PNG\r\n\x1a\n"
     image.RegisterFormat("png", pngHeader, Decode, DecodeConfig)
}
```

إن التأثير هو أن التطبيق يحتاج لإجراء استيراد فارغ فقط لحزمة الصيغة التي يحتاجها لجعل وظيفة image.Decode قادرة على فك تشفيرها.

تستخدم حزمة database/sql آلية مماثلة لتسمح للمستخدمين بأن يثبّتوا تعريفات قاعدة البيانات التي يحتاجونها. كمثال:

```
import (
     "database/mysql"
     _ "github.com/lib/pq"               // enable support for Postgres
     _ "github.com/go-sql-driver/mysql"  // enable support for MySQL
)

db, err = sql.Open("postgres", dbname)     // OK
db, err = sql.Open("mysql", dbname)        // OK
db, err = sql.Open("sqlite3", dbname)      // returns error:
                                                unknown driver"sqlite3"
```

تمرين 10.1: مدّد برنامج jpeg بحيث يحوّل أي صيغة إدخال مدعومة إلى أي صيغة إخراج مطلوبة، باستخدام image.Decode لكشف صيغة المُدخلة، وعَلَم لاختيار صيغة المُخرج.

تمرين 10.2: عرّف وظيفة قراءة ملف أرشيف عامة قادرة على قراءة ملفات ZIP (archive/zip) وملفات POSIX tar (archive/tar). استخدم آلية تسجيل مماثلة للتي وصفناها أعلاه بحيث يمكن إضافة دعم كل صيغة من صيغ الملفات عند استخدام الاستيرادات الفارغة.

10.6 الحِزَم والتسمية

سنقدم في هذا الجزء نصيحة حول كيفية إتباع تقاليد Go المميزة في تسمية الحِزَم وأعضاءها.

عند صُنع حزمة، اجعل اسمها قصيرًا، ولكن ليس أقصر من اللازم بشكل يجعلها غامضة أو غير مفهومة. إن الحَزم الأكثر استخدامًا في المكتبة القياسية هي التي اسمها bufio, bytes, flag, fmt, http, io, json, os, sort, sync, time. كن واضحا و صِف الأمور بوضوح قدر الإمكان. كمثال، لا تُسمّي حزمة utility باسم util عندما يكون هناك اسم مثل imageutil أو ioutil محدد ودقيق. تجنب اختيار أسماء حِزَم شائعة الاستخدام للمتغيرات المحلية ذات الصلة، أو يمكنك إجبار عملاء الحزمة على استخدام "إعادة تسمية الاستيرادات" كما هو الحال في حزمة path.

تأخذ أسماء الحزم عادة شكل منفرد، وتستخدم الحزم القياسية مثل bytes و errors و strings الجمع لتجنب إخفاء الأنواع المناظرة المُلعنة مسبقًا، أو لتجنب التضارب مع كلمة مفتاحية كما هو الحال مع go/types.

تجنّب أسماء الحِزَم التي لها دلالات أخرى بالفعل. كمثال، استخدمنا في الاسم اسم temp في حزمة تحويل درجة الحرارة في القسم 2.5، ولكن هذا الاسم لم يدُم لفترة طويلة، حيث كانت هذه فكرة سيئة جدًا لأن "temp" تكاد تكون مرادف عاليًا ل"temporary" أو مؤقّت. لقد مررنا بفترة موجزة استخدمنا في اسم temperature، ولكنه كان أطول من اللازم، ولم يعبّر عما تفعله الحزمة بالضبط. وفي النهاية، أصبح الاسم tempconv، وهو اسم أقصر وموازي لـ strcony.

لننتقل الآن إلى تسمية أعضاء الحِزَم. نظرًا لكون كل إشارة إلى عضو في حزمة أخرى يستخدم مُعرّف مؤهل مثل fmt.Println، فإن عبء وصف عضو الحزمة يتحمله اسم الحزمة ورقم العضو. نحن لسنا بحاجة لذكِر مفهوم التهيئة في Println لأن اسم الحزمة fmt يقوم بهذا بالفعل. عند تصميم حزمة، فكر في كيفية عمل الجزأين(المعرف المؤهل واسم العضو) معا وليس اسم العضو وحده. إليك أمثلة على بعض الخصائص المميزة:

```
bytes.Equal          flag.Int          http.Get
json.Marshal
```

يمكننا تحديد بعض أسماء الأنماط الشائعة. وتقدم حزمة strings عدد من الوظائف المستقلة للتلاعب بالسلاسل:

```
package strings
```

```
func Index(needle, haystack string) int
type Replacer struct{ /* ... */ }
func NewReplacer(oldnew ...string) *Replacer
type Reader struct{ /* ... */ }
func NewReader(s string) *Reader
```

لا تظهر كلمة string في أي من أسماءهم، ويشير العملاء لهم بـ strings.Index و strings.Replacer إلخ.

إن الحِزَم الأخرى التي يمكننا وصفها بـ "حزم من نوع واحد" (single-type packages) مثل html/template أو math/rand، تكشف عن نوع بيانات واحد أساسي بالإضافة إلى طُرقه، وعادة عن وظيفة New لصُنع حالات منها .

```
package rand // "math/rand"

type Rand struct{ /* ... */ }
func New(source Source) *Rand
```

يمكن أن يؤدّي هذا إلى تكرار، كما هو الحال في template.Template أو rand.Rand، ولهذا السبب عادة ما تكون أسماء هذا النوع من الحِزم قصيرة بشكل خاص.

على النقيض، هناك حِزم مثل net/http التي بها الكثير من الأسماء التي بدون بِنية كبيرة لأنهم يؤدون مهمة معقدة. بالرغم من احتواء الحزمة على 20 نوع ووظائف أكثر بكثير، إلا أن أهم أعضاء الحزمة أسماءهم بسيطة جدًا: Get, Post, Handle, Error, Client, Server.

10.7 أداة Go

يدور بقية الفصل حول أداة go، والتي تُستخدم في التحميل والاستعلام والتهيئة والبناء والاختبار وتثبيت حزم الشفرة المصدرية.

تدمج أداة go بين خصائص مجموعة متنوعة من الأدوات في مجموعة أوامر واحدة. وهي مدير حزمة (مناظر لـ apt أو rpm) الذي يجيب على الاستعلامات حول فهرس الحزمة، ويحسب اعتمادياتهم، ويحملهم من نظم تحكم عن بُعد في الإصدار. وهو نظام بناء يحسب اعتماديات الفصل ويحفّز المُجمّعات والمترجمات والروابط، بالرغم من أنه أقل اكتمالاً مبدئيًا من Unix make القياسي، كما سنرى في الفصل 11.

تستخدم واجهة سطر الأوامر الخاصة به طراز "سكين الجيش السويسري" (Swiss army knife)، الذي يحتوي على أكثر من دستة أوامر فرعية، بعضها رأيناه بالفعل، مثل get, run, build, fmt. يمكنك تشغيل go help لترى فهم التوثيق المُدمَج فيه، ولكن في حالة المراجع، أدرجنا أكثر الأوامر شائعة الاستخدام فيها أدناه:

```
 $ go
  ...
    build       compile packages and dependencies remove object files
    clean       show documentation for package or symbol
    doc       print Go environment information
    env       run gofmt on package sources
    fmt       download and install packages and dependencies
    get        compile and install packages and dependencies
    list        list packages
    run       compile and run Go program
    test       test packages
    version     print Go version
    vet         run go tool vet on packages
Use "go help [command]" for more information about a command.
...
```

تعتمد أداة go بشدة على الاتفاقات للحفاظ على الحاجة إلى التشكيل في حدها الأدنى. على سبيل المثال، يمكن للأداء إيجاد حزمة التغليف الخاصة بها لو أُعطيت اسم ملف مصدر Go، ويمكن لهذه الأداة إيجاد حزمة تغليفها لأن كل دليل يحتوي على حزمة واحدة، ومسار استيراد الحزمة يتوافق مع التسلسل الهرمي للدليل في مساحة العمل. ولو منحت الأداة مسار استيراد الحزمة، ستتمكن من إيجاد الدليل المناظر التي تخزن فيه ملفات الكائن. كما يمكن أن تجد أيضًا URL الخادم الذي يستضيف مستودع شفرة المصدر.

10.7.1 تنظيم مساحة العمل

إن التشكيل الوحيد الذي يحتاجه معظم المستخدمين هو متغير بيئة GOPATH الذي يحدد جِذر مساحة العمل. عند الانتقال إلى مساحة عمل مختلفة، يُحدّث المستخدمون قيمة GOPATH. كمثال، نحدد GOPATH بـ HOME/gobook$ أثناء العمل على هذا الكتاب:

```
$ export GOPATH=$HOME/gobook
$ go get gopl.io/...
```

بعد أن تقوم بتحميل كل برامج هذا الكتاب باستخدام الأمر أعلاه، ستحتوي مساحة عملك على تسلسل هرمي كالتالي:

```
GOPATH/
    src/
         gopl.io/
              .git/
              ch1/
                   helloworld/
```

```
                    main.go
                dup/
                    main.go
                ...
            golang.org/x/net/
                .git/
                html/
                    parse.go
                    node.go
                    ...
    bin/
        helloworld
        dup
    pkg/
        darwin_amd64/
            ...
```

تحتوي GOPATH على ثلاثة أدلة فرعية، يحمل دليل src الفرعي شفرة المصدر، وتقيم كل حزمة في دليل اسمه هو مسار استيراد الحزمة بالنسبة $GOPATH/src، مثل gopl.io/ch1/helloworld. لاحظ أن مساحة عمل GOPATH واحدة تحتوي على مستودعات متعددة للتحكم في الإصدار تحت src، مثل gopl.io أو golang.org. إن الدليل الفرعي pkg هو المكان الذي تُخزّن فيه أدوات البناء الحزم المُجمّعة، ويحمل دليل bin الفرعي البرامج التنفيذي مثل helloworld.

يحدد أحد المتغيرات البيئية الثانية وهو GOROOT الدليل الجذري لتوزيعة Go، والذي يوفّر كل حزم المكتبة القياسية. تشبه بنية الدليل تحت GOROOT بنية GOPATH، لذا، كمثال، ملفات المصدر في حزمة fmt توجد في دليل $GOROOT/src/fmt. لا يحتاج المستخدمين أبدًا إلى ضبط GOROOT، لأن الوضع الاعتيادي هو أن أداة go ستستخدم الموقع المثبّت فيه المتغير.

يطبع أمر go env القيم الفعالة لمتغيرات البيئة المتصلة بسلسلة الأدوات، وهذا يتضمن القيم الاعتيادية للقيم المفقودة. يحدد GOOS نظام التشغيل المستهدف (كمثال: أندرويد، لينوكس، داروين، ويندوز)، ويحدد GOARCH بنية المعالجة المستهدف، مثل amd64 أو 386 أو arm. وبالرغم من أن GOPATH هو المتغير الذي يجب ضبطه، فإن المتغيرات الأخرى قد تظهر من حين لآخر في شروحاتنا.

```
$ go env
GOPATH="/home/gopher/gobook"
GOROOT="/usr/local/go"
GOARCH="amd64"
GOOS="darwin"
...
```

10.7.2 تنزيل الحِزَم

عند استخدام أداة go، لا يشير مسار استيراد الحزمة إلى أين يمكنك إيجادها في مساحة العمل المحلية فقط، بل يشير إلى أين تجدها في الانترنت، بحيث يمكن لـ go get من جلبها وتحديثها.

يمكن لأمر go get تحمل حزمة منفردة أو شجرة فرعية أو مستودع كاملين باستخدام علامة(...)، كما هو الحال في القسم السابق. يمكن للأداة حساب وتحميل كل اعتماديات الحزم المبدئية كذلك، ولهذا السبب ظهرت golang.org/x/net/html في مساحة العمل في المثال السابق.

بمجرد أن يقوم go get بتحميل الحزم، يقوم ببنائهم ثم يثبّت المكتبات والأوامر. سنلقي نظرة على تفاصيل الأمر في الجزء التالي، ولكن سترى في المثال مدى سهولة العملية. يحصل أول أمر أدناه على أداة golint، والتي تفحص مشكلات النمط الشائعة في شفرة مصدر Go. بينما يشغّل الأمر الثاني golint على gopl.io/ch2/popcount من القسم 2.6.2. وهو يقدم تقرير مفيد بأننا نسينا كتابة تعليق مستندي للحزمة:

```
$ go get github.com/golang/lint/golint
$ $GOPATH/bin/golint
gopl.io/ch2/popcount
src/gopl.io/ch2/popcount/main.go:1:1:
"Package popcount ..." تعليق الحزمة يجب أن يكون في شكل
```

يدعم أمر go get مواقع استضافة الشفرة ذات الشعبية مثل GitHub و Bitbucket و Launchpad، ويمكنه تقديم الطلبات المناسبة لنظمهم المُتحكمة في الإصدار. أما بالنسبة للمواقع الأقل شهرة، قد تضطر لتوضيح بروتوكول التحكم في الإصدار المُستخدم في مسار الاستيراد، مثل Git أو Mercurial. قم بتشغيل مسار استيراد go help للحصول على التفاصيل.

إن الأدلة التي يخلقها go get هي عملاء حقيقيين من مستودع بعيد، وليس مجرد نسخ ملفات، لذا يمكنك استخدام أوامر التحكم في الإصدار لرؤية مقارنة للنسخ المحلية التي صنعتها أو التي حدثتها في مراجعة مختلفة. كمثال، دليل golang.org/x/net هو عميل Git:

```
$ cd $GOPATH/src/golang.org/x/net
$ git remote –v
origin https://go.googlesource.com/net (fetch)
origin https://go.googlesource.com/net (push)
```

لاحظ أن اسم النطاق الواضح في مسار استيراد الحزمة، golang.org، يختلف عن اسم النطاق الفعلي في خادم جت ، go.googlesource.com. إن هذه خاصية في أداة go تجعل الحزم تستخدم اسم نطاق مخصص

في مسار استيرادها بينما هي مستضافة بواسطة خدمة عامة مثل googlesource.com أو github.com. تتضمن صفحات HTML التي تحت https://golang.org/x/net/html البيانات الوصفية الموضحة أدناه، والتي تعيد توجيه أداة go إلى مستودع Git في موقع الاستضافة الفعلي:

```
$ go build gopl.io/chl/fetch
$ ./fetch https://golang.org/xZnet/html | grep go-import
<meta name=”go-import”
content=”golang.org/x/net git https://go.googlesource.com/net”>
```

لو حددت عَلَم u-، فإن go get ستضمن أن كل الحزم التي تزورها، بما في ذلك الاعتماديات، مُحدّثة لأحدث نسخة لها قبل أن تُبنى وتُثبّت. وبدون هذا العَلَم، لن تُحدّث الحزم الموجودة محليًا بالفعل.

يستعيد أمر go get -u بشكل عام أحدث نسخة من كل حزمة، وهو أمر مريح عندما تبدأ، ولكنه قد يكون غير مناسب عند استخدامه في المشاريع المُوزّعة، حيث التحكم الدقيق في الاعتماديات ضروري لصحة الإصدار. إن الحل المعتاد لهذه المشكلة هو تزويد (vendor) الشفرة، بمعنى، صُنع نسخة محلية دائمة من كل الاعتماديات الضرورية، وتحديث هذه النسخة بعناية وعلى مهل. قبل Go 1.5، كان هذا يتطلب تغيير مسارات استيراد تلك الحزم، بحيث تتحول نسخة golang.org/x/net/html إلى gopl.io/vendor/golang.org/x/net/html. تدعم النسخ الأحدث من أداة go التزويد مباشرة، بالرغم من أننا لا نمتلك مساحة كافية في هذا الفصل للحديث عن الأمر بالتفصيل. انظر "أدلة التزويد- Vendor Directories" في ناتج أمر go help gopath.

تمرين 10.3: باستخدام fetch http://gopl.io/ch1/helloworld?go-get=1، اكتشف ما هي الخدمة التي تستضيف عينات شفرة هذا الكتاب. (طلبات HTTP من go get تتضمن مؤشر go-get بحيث تتمكن الخوادم من تمييزها عن طلبات المتصفح العادي).

10.7.3 بناء الحِزَم

يقوم أمر go build بترجمة كل حزمة معطيات. لو كانت النتيجة مكتبة، يتم حذفها، فالأمر يتأكد وحسب من أن الحزمة خالية من أخطاء التفسير. لو كانت الحزمة مُسماة حزمة رئيسية (main)، فإن go build يستدعي الرابط لصنع ملف تنفيذي في الدليل الحالي، واسم الملف التنفيذي سيكون مأخوذ من آخر قطاع في مسار استيراد الحزمة.

نظرًا لكون كل دليل يحتوي على حزمة واحدة، فإن كل برنامج تنفيذي، أو أمر بلغة يونيكس، سيحتاج لدليله الخاص. قد تكون هذه الأدلة أحيانًا أدلة متفرعة من الدليل الذي يحمل اسم cmd، مثل أمر golang.org/x/tools/cmd/godoc الذي يخدم توثيق حزمة Go من خلال واجهة الويب (انظر 10.7.4).

يمكن تحديد الحِزم بواسطة مسارات استيرادها، كما رأينا أعلاه، أو من خلال اسم دليل نسبي، يجب أن يبدأ بـمقطع (.) أو (..) حتى لو كان هذا غير مطلوب في المعتاد. لو لم يُقدّم أي معطيات، يتم افتراض الدليل الحالي. من ثم، تبني الأوامر التالية نفس الحزمة، وإن كان كل منها يكتب أمر تنفيذي في الدليل الذي يشغله go build:

```
$ cd $GOPATH/src/gopl.io/ch1/helloworld
$ go build
```

و

```
$ cd anywhere
$ go build gopl.io/ch1/helloworld
```

و

```
$ cd $GOPATH
$ go build ./src/gopl.io/ch1/helloworld
```

ولكن ليس:

```
$ cd $GOPATH
$ go build src/gopl.io/ch1/helloworld
Error: cannot find package "src/gopl.io/ch1/helloworld".
```

يمكن تحديد الحِزَم أيضًا كقائمة بأسماء الملف، بالرغم من أن هذا يُستخدم فقط عادة في البرامج الصغيرة والتجارب لمرة واحدة. لو كان اسم الحزمة main، فإن الاسم التنفيذي سينبع من الاسم الأساسي لأول ملف .go.

```
$ cat quoteargs.go
package main
import (
    "fmt"
    "os"
)
func main() {
    fmt.Printf("%q\n", os.Args[1:])
```

```
}
$ go build quoteargs.go
$ ./quoteargs one "two three" four\ five
["one" "two three" "four five"]
```

أما بالنسبة للبرامج التي يتم التخلص منها سريعًا كهذا البرنامج، فسنريد تشغيل الملف التنفيذي بمجرد بناءه. ويدمج أمر go run هاتين الخطوتين:

```
$ go run quoteargs.go one "two three" four\ five
["one" "two three" "four five"]
```

إن المُعطى الأول الذي لا ينتهي بـ .go يُفترض أنه هو بداية قائمة المعاملات التي تُدخل في ملف Go التنفيذي.

يبني أمر go build في العادة الحزمة المطلوبة وكل اعتمادياتها، ثم يرمي كل الشفرة المُفسّرة باستثناء الملف التنفيذي الأخير، لو وُجِد. إن كل من تحليل الاعتمادية والتفسير سريعين بشكل يثير الدهشة، ولكن مع نمو المشاريع إلى عشرات الحزم ومئات الآلاف من أسطُر الشفرة، قد يصبح الوقت المطلوب لإعادة تفسير الاعتماديات كبير، عدّة ثوان غالبًا، حتى لو لم تتغير هذه الاعتماديات على الإطلاق.

إن أمر go install مشابه جدًا لأمر go build، باستثناء أنه يحفظ الشفرة المترجَمة لكل حزمة وأمر بدلاً من إلقاءها بعيدًا. إن تُحفظ الحِزم المُتَرجمة تحت دليل $GOPATH/pkg المُناظِر لدليل src الذي يتواجد فيه المصدر، وتُحفظ الملفات التنفيذية للأمر في دليل $GOPATH/bin. (يضع العديد من المستخدمين $GOPATH/bin في مسار بحثهم التنفيذي). بعد ذلك، لا يشغل go build و go install المُترجم لهذه الحزم والأوامر لو لم يتغيروا، مما يجعل البناءات التالية أسرع بكثير. ولمزيد من الراحة، يقوم go build –i بتثبيت الحزم التي تُعتبر اعتماديات لهدف البناء.

نظرًا لتفاوت الحزم المترجمة حسب المنصة والبنية، يحفظهم go install تحت الدليل الفرعي الذي يدمج اسمه قيم GOOS ومتغيرات بيئة GOARCH. على سبيل المثال، على نظام تشغيل مالك، تُترجَم حزمة golang.org/x/net/html package وتثبّت في ملف golang.org/x/net/html.a تحت $GOPATH/pkg/darwin_amd64.

سيكون من البسيط والواضح إجراء "ترجمة متعددة" (cross-compile) لبرنامج Go، بمعنى، بناء ملف تنفيذي لنظم التشغيل المختلفة أو CPU المختلفة. اضبط قيمة متغيرات GOOS و GOARCH خلال البناء. يطبع برنامج cross اسم نظام التشغيل والمعمارية التي بني لها:

```
gopl.io/ch10/cross
```

```
func main() {
    fmt.Println(runtime.GOOS, runtime.GOARCH)
}
```

يُنتِج الأمر التالي ملفات تنفيذية 64 بِت و 32 بِت بالترتيب:

```
$ go build gopl.io/ch10/cross
$ ./cross
darwin amd64
$ GOARCH=386 go build gopl.io/ch10/cross
$ ./cross
darwin 386
```

قد تحتاج بعض الحزم إلى ترجمة نسخ مختلفة من الشفرة لمنصات أو معالِجات مختلفة للتعامل مع مشكلات قابلية النقل المنخفضة أو لتقديم نسخ محسنة من الروتينات المهمة كمثال لا للحصر. لو كان اسم الملف يتضمن نظام تشغيل أو بنية المعالج، كمثال، اسم كـ net_linux.go أو asm_amd64.s، فإن أداة go ستترجم الملف فقط عند البناء لهذا الهدف. هناك تعليقات خاصة اسمها build tags تمنح ترجمة أدق من هذا. كمثال، لو كان الملف يحتوي على هذا التعليق:

```
// +build linux darwin
```

قبل إعلان الحزمة (وأمر doc الخاص بها)، فإن go build سترجمه فقط عند البناء لنظام تشغيل لينوكس أو ماك OS X، ويقول هذا الأمر ألا تترجم الملف أبدا:

```
// +build ignore
```

لمزيد من التفاصيل، انظر قسم "قيود البناء -Build Constraints" في توثيق حزمة go/bild:

```
$ go doc go/build
```

10.7.4 توثيق الحِزَم

يشجع أسلوب Go بقوة على التوثيق الجيد لدوال الحزمة، وكل إعلان عن عضو الحزمة المُصدّر، وإعلان الحزمة نفسه يجب أن يسبقه فورًا تعليق يشرح هدفه واستخدامه.

إن تعليقات doc في Go دائمًا ما تكون جُمل كاملة، والجملة الأولى عادة ما تكون ملخص يبدأ بالاسم المُعلن. تُذكر معاملات الوظيفة والمُعرّفات الأخرى بدون اقتباس أو رقم. كمثال، إليك تعليق doc لـ fmt.Fprintf.

```
// Fprintf formats according to a format specifier and writes to w.
```

```
// It returns the number of bytes written and any write error
// encountered.
func Fprintf(w io.Writer, format string, a ...interface{}) (int, error)
```

تُشرح تفاصيل تهيئة Fprintf في تعليق doc المرتبط بحزمة fmt نفسها. ويُعتبر التعليق السابق لإعلان الحزمة مباشرة هو تعليق doc الخاص بالحزمة ككل. يجب أن يكون هناك تعليق واحد فقط، وإن كان يمكن أن يظهر في أي ملف. قد تستحق تعليقات الحزمة الأطول ملف خاص بها، وعندما يزيد عدد سطور fmt عن 300 سطر، يُطلق على الملف عادة doc.go.

لا يجب أن يكون التوثيق الجيد مسهبا، والتوثيق ليس بديل للبساطة. إن العادات المتفق عليها في Go هي الاختصار والبساطة في التوثيق كما هو الحال في كل الأشياء، حيث أن التوثيق، مثله مثل الشفرة، يحتاج لصيانة كذلك. يمكن شرح العديد من الإعلانات في جملة واحدة ذات صياغة جيدة، ولو كان السلوك واضح حقًا، قد لا نحتاج للتعليقات على الإطلاق.

لقد قدمنا في هذا الكتاب – حسبما سمحت المساحة – العديد من الإعلانات التي يسبقها تعليقات doc، ولكنك ستجد أمثلة أفضل بينما تتصفح المكتبة القياسية. يمكن لأداتين مساعدتك على فعل هذا.

تطبع أداة go doc الإعلان وتعليق doc الخاص بالكيان المحدد في سطر الأمر، والذي قد يكون حزمة:

```
$ go doc time
package time // import "time"
```

تقدم حزمة time وظيفة لقياس وعرض الوقت.

```
const Nanosecond Duration = 1 ...
func After(d Duration) <-chan Time
func Sleep(d Duration)
func Since(t Time) Duration
func Now() Time
type Duration int64
type Time struct { ... }
...many more...
```

أو عضو الحزمة:

```
$ go doc time.Since
```

تعيد Since الوقت الذي مر منذ t
تُختصر إلى: time.Now().Sub(t)

أو طريقة:

```
$ go doc time.Duration.Seconds
func (d Duration) Seconds() float64
	تُعيد الثواني المُدة كرقم فاصلة عائمة بالثواني.
```

لا تحتاج الأداة إلى مسارات استيراد كاملة أو حالة مُعرّف صحيح، ويطبع هذا الأمر توثيق (*json.Decoder). Decode من حزمة encoding/json:

```
$ go doc json.decode
func (dec *Decoder) Decode(v interface{}) error
```

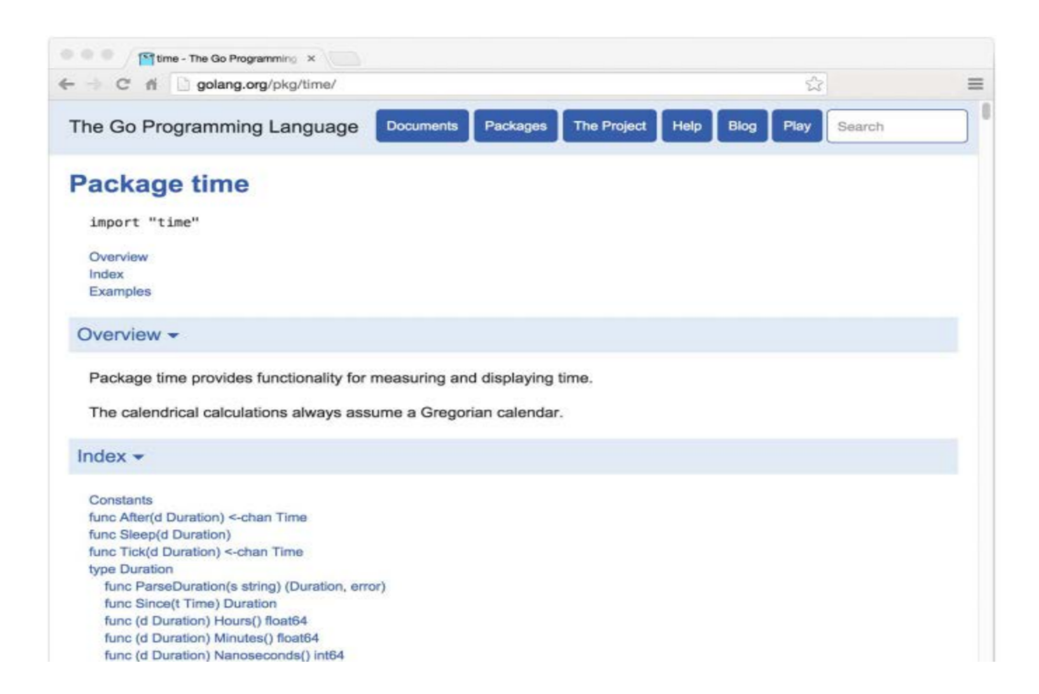
تقرأ Decode قيمة JSON المشفرة التالية من مُدخلاتها وتخزينها في القيمة التي تشير لها v.

إن الأداة الثانية، ذات الاسم المُربِك godoc، تعمل كصفحات HTML ذات روابط متقاطعة تقدم نفس المعلومات التي يقدمها go doc والكثير غيرها. يغطي خادم godoc على https://golang.org/pkg المكتبة القياسية. ويوضح الشكل 10.1 توثيق حزمة time، وسنرى في الجزء في 11.6 عرض تفاعلي لـ godoc لبرامج كأمثلة. يحتوي خادم godoc على https://godoc.org فهرس قابل للبحث يحتوي على آلاف الحزم مفتوحة المصدر.

يمكنك تشغيل نسخة من godoc في مساحة عملك لو أردت تصفح حِزَمك الخاصة. زُر http://localhost:8000/pkg في متصفحك بينما تشغيل هذا الأمر:

```
$ godoc -http :8000
```

إن أعلام -analysis=type و -analysis=pointer الخاصة بها تدعم التوثيق وشفرة المصدر بنتائج تحليل ساكن متقدم.



الشكل 10.1 توثيق حزمة time

10.7.5 الحِزم الداخلية

إن الحزمة هي الآلية الأكثر أهمية للتغليف في برامج Go. وتظهر المُعرّفات غير المُصدّرة داخل نفس الحزمة فقط، بينما أن المُعرّفات المُصدّرة تظهر للعالم.

أحيانًا قد يكون الحل الوسط مفيد بشكل أكثر، وهو طريقة لتحديد المُعرّفات الظاهرة لمجموعة صغيرة من الحزم الموثوق فيها، ولكنها لا تظهر للجميع. كمثال، عندما تُقسّم حزمة كبيرة إلى أجزاء أصغر للعمل عليها بشكل أسهل، قد لا نرغب في الكشف عن الواجهات بين تلك الأجزاء أمام الحزم الأخرى، أو قد نرغب في مشاركة وظائف الوسائل والمنافع عبر حزم متعددة في مشروع بدون كشفهم على نطاق أوسع. أو ربما نرغب وحسب في تجربة حزمة جديدة بدون الالتزام بـ API الخاص بها قبل الأوان، فنضعها "تحت المراقبة" مع مجموعة محدودة من العملاء.

حتى تتمكن الحزمة من التعامل مع هذه الاحتياجات، تقوم أداة go buils بمعاملة الحزمة بطريقة خاصة لو كان مسار استيرادها يحتوي على مقطع مسار اسمه internal. ويُطلق على هذه الحزم اسم الحزم الداخلية (internal packages). يمكن استيراد الحزمة الداخلية فقط بواسطة حزمة أخرى توجد داخل شجرة متجذرة كدليل أم للدليل الداخلي. كمثال، لو نظرنا للحزم أدناه، سنجد أن بإمكاننا استيراد net/http/internal/chund مِن net/http/httputil أو net/http ولكن ليس من net/url، ولكن يمكن لـ net/url استيراد net/http/httputil.

```
net/http
net/http/internal/chunked
```

```
net/http/httputil
net/url
```

10.7.6 استعلام الحِزَم

تقدم أداة go list تقريرا بالمعلومات الموجودة حول الحزم المتاحة. وتختر go list في أبسط أشكالها ما إذا كانت الحزمة موجودة في مساحة العمل أم لا، وتطبع مسار استيرادها لو كانت موجودة:

```
$ go list github.com/go-sql-driver/mysql
github.com/go-sql-driver/mysql
```

قد يحتوي معطى go list على حرف بدل (...) يمكن أن يطابق أي سلسلة فرعية في مسار استيراد الحزمة. ويمكننا استخدامها لتعديد كل الحزم داخل مساحة عمل Go:

```
$ go list ...
archive/tar
archive/zip
bufio
bytes
cmd/addr2line
cmd/api
. ..many more...
```

أو داخل شجرة فرعية محدد:

```
$ go list gopl.io/ch3/...
gopl.io/ch3/basename1
gopl.io/ch3/basename2
gopl.io/ch3/comma
gopl.io/ch3/mandelbrot
gopl.io/ch3/netflag
gopl.io/ch3/printints
gopl.io/ch3/surface
```

أو مرتبط بموضوع معين:

```
$ go list ...xml...
encoding/xml
gopl.io/ch7/xmlselect
```

يحصل أمر go list على بيانات وصفية كاملة لكل حزمة، وليس مسار الاستيراد فقط، ويتيح المعلومات للمستخدمين أو الأدوات الأخرى بصيغ متعددة. يجعل عَلَم json– أمر go list يطبع السجل الكامل لكل حزمة في صيغة JSON:

```
$ go list -json hash
{
    "Dir": "/home/gopher/go/src/hash",
    "ImportPath": "hash",
    "Name": "hash",
    "Doc": "Package hash provides interfaces for hash functions.",
    "Target": "/home/gopher/go/pkg/darwin_amd64/hash.a",
    "Goroot": true,
    "Standard": true,
    "Root": "/home/gopher/go",
    "GoFiles": [
          "hash.go"
    ],
    "Imports": [
          "io"
    ],
    "Deps": [
          "errors",
          "io",
          "runtime",
          "sync",
          "sync/atomic",
          "unsafe"
    ]
}
```

يتيح عَلَم f– للمستخدمين إمكانية تخصيص صيغة الناتج باستخدام قالب لغة في الحزمة text/template (انظر 4.6). يطبع هذا الأمر الاعتماديات المتعدية لحزمة strconv، ويفصل بينها بمسافات:

```
$ go list -f '{{join .Deps " "}}' strconv
errors math runtime unicode/utf8 unsafe
```

ويطبع هذا الأمر الاستيرادات المباشرة لكل حزمة في شجرة فرعية مضغوطة في المكتبة القياسية:

```
$ go list -f '{{.ImportPath}} -> {{join .Imports " "}}' compress/...
compress/bzip2 -> bufio io sort
compress/flate -> bufio fmt io math sort strconv
compress/gzip -> bufio compress/flate errors fmt hash hash/crc32 io time
compress/lzw -> bufio errors fmt io
compress/zlib -> bufio compress/flate errors fmt hash hash/adler32 io
```

إن أمر go list مفيد بالنسبة لكل من الاستعلامات التفاعلية لمرة واحدة، وفي بناء واختبار نصوص الأتمتة. سنستخدمه مرة أخرى في القسم 11.2.4. لمزيد من المعلومات حول مجموعة الحقول المتاحة ومعانيها، انظر ناتج go help list.

لقد شرحنا في هذا الفصل كل الأوامر الفرعية المهمة في أداة go باستثناء أمر واحد. سنرى في الفصل التالي كيف يُستخدم أمر go test لاختبار برامج Go.

تمرين 10.4: قم ببناء أداة تقدم تقريرا بإعدادات كل الحزم في مساحة العمل واجعلها تعتمد بشكل متعدّي على الحزم التي تحددها المعطيات. تلميح: ستحتاج لتشغيل go list مرتين، مرة في الحزم المبدئية ومرة لكل الحزم. قد ترغب في تحليل ناتج JSON الخاص به باستخدام حزمة encoding/json (انظر 4.5).

11- الاختبار – Testing

إن موريس ويلكس (Maurice Wilkes) مُطوِّر EDSAC، أول كمبيوتر ذو برامج مُخزّنة فيه، وصل لفكرة مذهلة أثناء صعوده سلم معمله عام 1949. وقد قال في كتابه "مذكرات رائد كمبيوتر": "لقد أدركت فجأة ودون سابق إنذار أنني سأقضي جزءا كبير من حياتي الباقي في اكتشاف الأخطاء في برامجي الخاصة". ويمكن لكل مبرمج لكمبيوتر ببرامج مُخزنة منذ ذلك الحين أن يتعاطف مع ويلكس، وإن كانوا سيتعجبون قليلاً من سذاجته بشأن صعوبات بناء البرمجيات.

إن البرامج اليوم أصبحت أكبر وأكثر تعقيدًا بكثير عما كانت عليه أيام ويلكس بالطبع، ويُبذل الكثير من الجهود على وضع تقنيات لتسهيل التعامل مع هذا التعقيد. هناك تقنيتان بارزتان بشكل خاص نظرًا لفعاليتهما الشديدة في هذه المهمة. الأولى هي مراجعة الزملاء الروتينية للبرامج قبل نشرها، والثاني هي موضوع هذا الفصل، وهي الاختبار.

نحن نعني بالاختبار "الاختبار الآلي - automated testing"، وهو كتابة برامج صغيرة تفحص الشفرة قيد الاختبار (شفرة الإنتاج – Production Code)، وتتصرف كما هو متوقع مع مُدخلات معينة، والتي عادة ما تختار بحرص لتفعيل خصائص معينة أو اختيارها بشكل عشوائي لضمان أوسع تغطية ممكن لكل الاحتمالات.

إن مجال اختبار البرمجيات ضخم جدًا، وتشغل مهمة الاختبار كل المبرمجين في أوقات كثيرًا، وأحيانًا تشغل بعض المبرمجين طوال الوقت. تتضمن البحوث المنشورة حول الاختبار آلاف الكتب المطبوعة وملايين الكلمات في منشورات المدونات. هناك عشرات حزم البرمجيات في كل لغة برمجة معروفة هدفها هو اختبار بناء البرامج، وبعضها يحتوي على كم كبير من النظريات، ويبدو أن هذا المجال يجذب الكثير من المحترفين ذوي المتابعين الأشبه بطائفة خاصة بهم. ويكفي القول أن بإمكانك إقناع المبرمجين أن عليهم اكتساب مجموعة مهارات كاملة جديدة حتى يتمكنوا من كتابة اختبارات فعالة.

إن منهج Go في الاختبار يبدو منخفض التقنية بالمقارنة، فهو يعتمد على أمر واحد هو go test، ومجموعة من الاتفاقات الاصطلاحية (conventions) لكتابة وظائف الاختبار التي يمكن لـ go test تشغيلها. إن الآلية الخفيفة نسبيًا فعالة في الاختبار الخالص، وهي تمتد بطبيعتها إلى قياسات الأداء (benchmarks) وأمثلة التوثيق المنهجية.

من الناحية العملية، لا تختلف كتابة شفرة اختبار كثيرًا عن كتابة البرنامج الأصلي نفسه. نحن نكتب وظائف قصيرة تركز على جزء واحد من المهام، ويجب أن نكون حذرين في وضع شروط الحدود، وأن نفكر في بِنيات البيانات، وأن نفكر منطقيًا في النتائج التي يجب أن تنتج عن المُدخلات المناسبة، وهذه هي نفس عملية كتابة شفرة Go عادية، ولا تحتاج لتدوينات أو اصطلاحات أو أدوات جديدة.

11.1 أداة go test

إن أمر go test الفرعي هو مشغل الاختبارات لحزم Go التي تُنظم وفقًا لاتفاقات اصطلاحية معينة. وفي دليل الحزمة، لا تُعد الملفات التي تنتهي أسماءها بـ test.go_ جزءا من الحزمة التي بُنيت عادة بـ go build ولكنها تصبح جزء منها عندما تُبنى بـ go test.

هناك ثلاثة أنواع من الوظائف في ملفات test.go_* تُعامل معاملة خاصة، وهي الاختبارات وقياسات الأداء والأمثلة. إن "وظيفة الاختبار" (test function)، وهي الوظيفة التي يبدأ اسمها بـ Test، تقوم بنوع من أنواع المنطق في البرنامج لتصحيح السلوك، وتستدعي go test وظيفة الاختبار وتقدم النتائج، والتي إما أن تكون نجاح (PASS) أو فشل (FAIL). أما "وظيفة أداة القياس" (benchmark function)، فهي التي يبدأ اسمها بـ Benchmark، وتقيس أداء بعض العمليات، وتقدم go test تقرير بمتوسط وقت تنفيذ العملية. و "وظيفة المثال" (example function)، والتي يبدأ اسمها بـ Example، فتقدم توثيق مفحوص بواسطة آلة. سنغطي الاختبارات بالتفصيل في القسم 11.2، وقياسات الأداء في القسم 11.4، والأمثلة في القسم 11.6.

تفحص أداة go test ملفات test.go_* وتبحث عن وظائف الخاصة، وتنشيء حزمة رئيسية مؤقتة تستدعيهم كلهم بطريقة سليمة، وتبنيهم وتشغلهم، وتقدم النتائج، ثم تنظف كل شيء.

11.2 وظائف Test

يجب أن يستورد كل ملف test حزمة اختبار، وتحتوي وظائف الاختبار على التوقيع التالي:

```
func TestName(t *testing.T) {
    // ...
}
```

يجب أن تبدأ أسماء وظيفة الاختبار بـ Test، واللاحقة الاختيارية Name يجب أن تبدأ بحرف كبير:

```
func TestSin(t  *testing.T)     {     /*    ...    */    }
func TestCos(t  *testing.T)     {     /*    ...    */    }
func TestLog(t  *testing.T)     {     /*    ...    */    }
```

يقدّم المؤشر t أساليب لتقديم تقرير بإخفاقات الاختبار، ويسجل معلومات إضافية. لنُعرف حزمة المثال: gopl.io/ch11/word1، تحتوي الحزمة على وظيفة منفردة هي IsPalindrome والتي تقدم تقرير بما إذا كانت السلسلة

تقرأ نفس الكلام للأمام والخلف. (يختبر هذا التطبيق كل بايت مرتين لو كانت السلسلة palindrome، وسنعود لهذا لاحقًا).

```
gopl.io/ch11/word1

// Package word provides utilities for word games.
package word
// IsPalindrome reports whether s reads the same forward and backward.
// (Our first attempt.)
func IsPalindrome(s string) bool {
    for i := range s {
        if s[i] != s[len(s)-1-i] {
            return false
        }
    }
    return true
}
```

في نفس المسار، يحتوي الملف word_test.go على وظيفتيّ اختبار اسمهما TestPalindrome و TestNonPalindrome. وتتأكد كل منهما من أن IsPalindrome يقدم الإجابة الصحيحة لمُدخَل واحد ويذكر الإخفاقات باستخدام t.Error:

```
package word
import "testing"
func TestPalindrome(t *testing.T) {
    if !IsPalindrome("detartrated") {
        t.Error(`IsPalindrome("detartrated") = false`)
    }
    if !IsPalindrome("kayak") {
        t.Error(`IsPalindrome("kayak") = false`)
    }
}
func TestNonPalindrome(t *testing.T) {
    if IsPalindrome("palindrome") {
        t.Error(`IsPalindrome("palindrome") = true`)
    }
}
```

إن أمر go test (أو go build) بدون معطيات حزمة يعمل في الحزمة في الدليل الحالي. ويمكننا بناء وتشغيل الاختبارات من خلال الأمر التالي:

```
$ cd $GOPATH/src/gopl.io/ch11/word1
$ go test
ok gopl.io/ch11/word1 0.008s
```

وهذا مُرضي، وبالتالي يجعلنا نشحن البرنامج، ولكن بمجرد انتهاء إطلاق البرنامج، تبدأ تقارير الخلل في الظهور. اشتكى مستخدم فرنسي اسمه Noelle Eve Elleon من أن IsPalindrome لا يتعرف على "été"، بينما قال مستخدم آخر من

أمريكا الوسطى أنه خائب الأمل من أن البرنامج يرفض عبارة "A man, a plan, a canal: Panama.". إن تقارير الخلل الصغيرة والمحددة هذه تحتاج لاختبارات جديدة بطبيعة الحال.

```
func TestFrenchPalindrome(t *testing.T) {
    if !IsPalindrome("été") {
        t.Error(`IsPalindrome("été") = false`)
    }
}
func TestCanalPalindrome(t *testing.T) {
    input := "A man, a plan, a canal: Panama"
    if !IsPalindrome(input) {
        t.Errorf(`IsPalindrome(%q) = false`, input)
    }
}
```

نستخدم Errorf لتجنب كتابة سلسلة مُدخل طويلة مرتين، فهو يقدم تهيئة مثل Printf.

عند إضافة اختبارين جديدين، يفشل أمر go test في تقديم رسائل إخطار بالخلل.

```
$ go test
---FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("ete") = false
---FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
FAIL gopl.io/ch11/word1 0.014s
```

إن كتابة الاختبار أولاً وملاحظة ما يحفز نفس الفشل الذي وصفه تقرير الخلل الذي قدمه المستخدم هو أحد المناهج الجيدة في العمل، فحينها فقط يمكننا التأكد أن الحل الذي وجدناه يعالج المشكلة الصحيحة.

علاوة على ذلك، فإن أحد المميزات الإضافية هي أن تشغيل go test عادة ما يكون أسرع من مراجعة الخطوات المذكورة في تقرير الخلل يدويًا، مما يسمح لنا بتكرار الأمر بسرعة أكبر. لو كانت تركيبة الاختبار تحتوي على العديد من الاختبارات البطيئة، فيمكننا تحقيق تقدم أسرع لو كنا انتقائيين بشأن الاختبارات التي نجريها.

يطبع عَلَم v- الاسم ووقت التنفيذ الخاص بكل اختبار في الحزمة:

```
$ go test -v
=== RUN TestPalindrome
--- PASS: TestPalindrome (0.00s)
=== RUN TestNonPalindrome
--- PASS: TestNonPalindrome (0.00s)
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("ete") = false
=== RUN TestCanalPalindrome
```

```
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL gopl.io/ch11/word1 0.017s
```

وعلم run–، الذي كان المعطى الخاص به هو تعبير منتظم، يجعل go test يشغل الاختبارات التي تتطابق وظيفتها مع النمط فقط:

```
$ go test -v -run="French|Canal"
=== RUN TestFrenchPalindrome
--- FAIL: TestFrenchPalindrome (0.00s)
    word_test.go:28: IsPalindrome("ete") = false
=== RUN TestCanalPalindrome
--- FAIL: TestCanalPalindrome (0.00s)
    word_test.go:35: IsPalindrome("A man, a plan, a canal: Panama") = false
FAIL
exit status 1
FAIL gopl.io/ch11/word1 0.014s
```

بعد أن ننجح في الاختبارات المنتقاة، يجب أن نفعّل go test بدون أعلام ليشغل مجموعة الاختبار كلها مرة أخيرة قبل أن نلتزم بالتغيير.

إن مهمتنا الآن هي إصلاح الخلل، وكشف بحث سريع أن سبب الخلل الأول هو استخدام IsPalindrome لتسلسلات byte وليس تسلسلات rune، وبالتالي فإن الحروف التي ليست ASCII مثل ال e في "ete" تربكها. أما الحل الثاني فناتج عن عدم تجاهل المسافات، وعلامات الترقيم، وحالة الحروف.

بالتالي كتبنا الوظيفة بحرص أكثر:

```
gopl.io/ch11/word2

// Package word provides utilities for word games.
package word
import "unicode"
// IsPalindrome reports whether s reads the same forward and backward.
// Letter case is ignored, as are non-letters.
func IsPalindrome(s string) bool {
    var letters []rune
    for _, r := range s {
        if unicode.IsLetter(r) {
            letters = append(letters, unicode.ToLower(r))
        }
    }
    for i := range letters {
        if letters[i] != letters[len(letters)-1-i] {
```

```
            return false
        }
    }
    return true
}
```

كتبنا أيضًا مجموعة شاملة أكثر من حالات الاختبار التي تدمج كل الحالات السابقة مع عدد من الحالات الجديدة في جدول واحد.

```
func TestIsPalindrome(t *testing.T) {
    var tests = []struct {
        input string
        want  bool
    }{
        {"", true},
        {"a", true},
        {"aa", true},
        {"ab", false},
        {"kayak", true},
        {"detartrated", true},
        {"A man, a plan, a canal: Panama", true},
        {"Evil I did dwell; lewd did I live.", true},
        {"Able was I ere I saw Elba", true},
        {"été", true},
        {"Et se resservir, ivresse reste.", true},
        {"palindrome", false}, // non-palindrome
        {"desserts", false},   // semi-palindrome
    }
    for _, test := range tests {
        if got := IsPalindrome(test.input); got != test.want {
            t.Errorf("IsPalindrome(%q) = %v", test.input, got)
        }
    }
}
```

كل اختباراتنا الجديدة نجحت:

```
$ go test gopl.io/ch11/word2
ok  gopl.io/ch11/word2    0.015s
```

إن هذا النمط الجديد في الاختبار "المدفوع بالجدول" (table-driven) شائع جدًا في Go. وهو مباشر جدًا في إضافة مُدخلات جديدة في الجدول حسب الحاجة، وحيث أن منطق التأكيد لا يتكرر، يمكننا استثمار مزيد من الجهد في إنتاج رسالة خطأ جيدة.

إن نتيجة الاختبار الفاشل لا تتضمن تتبع الرصّة بأكملها في لحظة استخدام t.Errorf، ولا تسبب t.Errorf حالة هلع أو وقف لتنفيذ الاختبار، على العكس من حالات فشل التأكيد الموجودة في العديد من إطارات الاختبار الخاصة باللغات الأخرى. إن الاختبارات مستقلة عن بعضها، ولو كان هناك مُدخل مبكر للجدول يجعل الاختبار يفشل، فإن مُدخلات

الجدول اللاحقة سيتم فحصها برغم ذلك، وبالتالي يمكننا اكتشاف الكثير من الأمور حول الإخفاقات المتعددة خلال كل تشغيل منفرد.

عندما نضطر حقًا لإيقاف وظيفة اختبار، ربما بسبب فشل شفرة بدء أو لمنع خلل وقع بالفعل من التسلسل في سلسلة اختلالات في أشياء أخرى، نستخدم t.Fatal أو t.Fatalf. يجب أن نستدعيهم من نفس الـ goroutine الموجود فيه وظيفة الاختبار، وليس من وظيفة أخرى نشأت خلال الاختبار.

إن رسائل فشل الاختبار عادة ما تكون في الشكل "f(x) = y, want z"، حيث f(x) يفسر العملية التي تمت محاولة إجراءها ومُدخلها، و y هو النتيجة الفعلية، و z هو النتيجة المتوقعة. كلما أمكن – كما هو الحال في مثال palindrome – تُستخدم بِنية Go فعلية في الجزء الخاص بـ f(x). إن عرض x مهم بشكل خاص في الاختبار المدفوع بالجدول، حيث أن أي تأكيد مُعطى يُنفذ مرات عديدة بقيم مختلفة. تجنّب المعلومات النمطية والزائدة عن اللازم. عند اختبار وظيفة منطقية (Boolean) مثل IsPalindrome، احذف الجزء الخاص بـ z لأنه لا يضيف أي معلومات جديدة. لو كان x أو y أو z طوال، اطبع ملخصا دقيقا بالأجزاء المهمة بدلاً من ذلك. يجب على مؤلف الاختبار أن يسعى لمساعدة المبرمج الذي يجب أن يشخّص الفشل في الاختبار.

تمرين 11.1: اكتب اختبارات لبرنامج charcount الموجود في القسم 4.3.

تمرين 11.2: اكتب مجموعة من الاختبارات لـ IntSet (انظر 6.5) تتحقق من أن سلوكه بعد كل عملية يكافئ مجموعة قائمة على الخرائط المدمجة. احفظ تطبيقك لقياس أداء في التمرين 11.7.

11.2.1 الاختبار العشوائي – Randomized testing

إن الاختبارات المدفوعة بالجدول مناسبة للتحقق من أن الوظيفة تعمل على مُدخلات مختارة بعناية لاختبار حالات مثيرة للاهتمام في المنطق. هناك منهج آخر وهو "الاختبار العشوائي"، والذي يستكشف نطاق مُدخلات أوسع من خلال بناء مُدخلات عشوائيًا.

كيف نعلم ما المخرج المتوقّع من وظيفتنا لو أُدخل فيها مُدخل عشوائي؟ هناك استراتيجيتان. الأولى هي كتابة تطبيق بديل للوظيفة يستخدم خوارزمية أقل كفاءة ولكنها أبسط وأوضح، ويتأكد من أن كلا التطبيقين يقدم نفس النتيجة. الاستراتيجية الثانية هي إنشاء قيم منتجات وفقًا لنمط معين بحيث نعلم ما هو المُخرج المتوقع.

يستخدم المثال أدناه المنهج الثاني: تنتج وظيفة randomPalindrome كلمات معروف أنها palindromes وفقًا لطبيعة بنيتها.

```
import "math/rand"
```

```go
// randomPalindrome returns a palindrome whose length and contents
// are derived from the pseudo-random number generator rng.
func randomPalindrome(rng *rand.Rand) string {
    n := rng.Intn(25) // random length up to 24
    runes := make([]rune, n)
    for i := 0; i < (n+1)/2; i++ {
        r := rune(rng.Intn(0x1000)) // random rune up to '\u0999'
        runes[i] = r
        runes[n-1-i] = r
    }
    return string(runes)
}
func TestRandomPalindromes(t *testing.T) {
    // Initialize a pseudo-random number generator.
    seed := time.Now().UTC().UnixNano()
    t.Logf("Random seed: %d", seed)
    rng := rand.New(rand.NewSource(seed))
    for i := 0; i < 1000; i++ {
        p := randomPalindrome(rng)
        if !IsPalindrome(p) {
            t.Errorf("IsPalindrome(%q) = false", p)
        }
    }
}
```

نظرًا لكون الاختبارات العشوائية غير محددة، من الضرورة أن يحتوي سجل الاختبار الفاشل على معلومات كافية لاستنساخ هذا الفشل. وفي مثالنا، يخبرنا المُدخل p في IsPalindrome بكل ما نحتاج لمعرفته، ولكن في الوظائف التي تقبل مُدخلات أكثر تعقيدًا، قد يكون من الأبسط إدخال بذرة مولّد أرقام شبه عشوائي (كما فعلنا أعلاه) بدلاً من التخلي عن بنية بيانات المُدخلات بأكملها. إن وجود قيمة البذرة هذه يساعدها على تعديل الاختبار بسهولة للاستجابة للخلل بشكل محدد.

لو استخدمنا الوقت لحالي كمصدر للعشوائية، فإن الاختبار سيستكشف المُدخلات الجديدة في كل مرة يعمل فيها، على مدار عمره الافتراضي كله. هذا قيّم بشكل خاص لو استخدم مشروعك نظام آلي لإجراء كل الاختبارات دوريًا.

تمرين 11.3: إن TestRandomPalindromes يختبر الـ palindromes فقط، اكتب اختبار عشوائيا ينتج ويتحقق من صحة الـ non-palindromes.

تمرين 11.4: عدّل randomPalindrome للتمرين على تعامل IsPalindrome مع علامات الترقيم والمسافات.

11.2.2 اختبار أمر ما

إن أداة go test مفيدة في اختبار حزم المكتبة، ولكن بقليل من الجهد، يمكن استخدامها لاختبار الأوامر أيضًا. إن الحزمة المُسماة بحزمة main تنتج عنها برنامج تنفيذي، ولكن يمكن استيرادها كمكتبة أيضًا.

لنكتب اختبار لبرنامج الصدى (echo) الموجود في القسم 2.3.2، لقد قسمنا البرنامج إلى وظيفتين: echo تقوم بالعمل الحقيقي، بينما main توزع وتقرأ قيم العَلَم وتقدم تقرير بأي أخطاء يقدمها الصدى.

```go
gopl.io/ch11/echo
// Echo prints its command-line arguments.
package main
import (
    "flag"
    "fmt"
    "io"
    "os"
    "strings"
)
var (
    n = flag.Bool("n", false, "omit trailing newline")
    s = flag.String("s", " ", "separator")
)
var out io.Writer = os.Stdout // modified during testing
func main() {
    flag.Parse()
    if err := echo(!*n, *s, flag.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "echo: %v\n", err)
        os.Exit(1)
    }
}
func echo(newline bool, sep string, args []string) error {
    fmt.Fprint(out, strings.Join(args, sep))
    if newline {
        fmt.Fprintln(out)
    }
    return nil
}
```

سنستدعي في الاختبار "echo" بمجموعة متنوعة من المعطيات وأوضاع العلم، ونتحقق من طباعته للمُخرج الصحيح في كل حالة، وبالتالي أضفنا معاملات إلى "echo" لتقليل اعتماده على المتغيرات العالمية. كما قدمنا أيضا متغير out عالمي آخر، io.Writer الذي ستُكتب فيه النتائج. إن جعل echo يكتب عبر هذا المتغير، وليس بشكل مباشر على os.Stdout، يمكّن الاختبارات من أن تحل محل تطبيقات Writer المختلفة التي تسجل ما يُكتب لفحصه لاحقًا. هذا هو الاختبار، في الملف echo_test.go:

```go
package main
import (
    "bytes"
    "fmt"
    "testing"
)
func TestEcho(t *testing.T) {
    var tests = []struct {
        newline bool
        sep     string
        args    []string
```

```
		want    string
	}{
		{true, "", []string{}, "\n"},
		{false, "", []string{}, ""},
		{true, "\t", []string{"one", "two", "three"}, "one\ttwo\tthree\n"},
		{true, ",", []string{"a", "b", "c"}, "a,b,c\n"},
		{false, ":", []string{"1", "2", "3"}, "1:2:3"},
	}
	for _, test := range tests {
		descr := fmt.Sprintf("echo(%v, %q, %q)",
			test.newline, test.sep, test.args)
		out = new(bytes.Buffer) // captured output
		if err := echo(test.newline, test.sep, test.args); err != nil {
			t.Errorf("%s failed: %v", descr, err)
			continue
		}
		got := out.(*bytes.Buffer).String()
		if got != test.want {
			t.Errorf("%s = %q, want %q", descr, got, test.want)
		}
	}
}
```

لاحظ أن شفرة الاختبار موجودة في نفس حزمة شفرة الإنتاج. بالرغم من أن اسم الحزمة هو main، وتعرّف وظيفة أساسية، إلا أن أثناء الاختبار، تعمل هذه الحزمة كمكتبة تكشف وظيفة TestEcho لتعريف الاختبار، ويتم تجاهل وظيفتها الرئيسية.

لو نظمنا الاختبار كجدول، يمكننا إضافة حالات اختبار جديدة بسهولة، لنرى ما سيحدث عند فشل الاختبار، من خلال إضافة هذا السطر للجدول:

```
{true, ",", []string{"a", "b", "c"}, "a b c\n"}, // NOTE: wrong expectation! go
test prints
$ go test gopl.io/ch11/echo
--- FAIL: TestEcho (0.00s)
    echo_test.go:31: echo(true, ",", ["a" "b" "c"]) = "a,b,c", want "a b c\n"
FAIL
FAIL gopl.io/ch11/echo 0.006s
```

تصف رسالة الخطأ محاولة التشغيل (باستخدام بنية Go-like)، والسلوك الفعلي، والسلوك المتوقع، بهذا الترتيب. ومع رسالة خطأ معلوماتية كهذه، يمكنك الحصول على فكرة واضحة حول السبب الجذري قبل أن تحدد حتى شفرة المصدر الخاصة بالاختبار.

من المهم ألا تستدعي الشفرة التي يتم اختبارها log.Fatal أو os.Exit، حيث أنهما سيوقفان العملية تمامًا، ويجب معرفة أن استدعاء هذه الوظائف هو حق حصري لـ main. لو حدث شيء غير متوقع على الإطلاق، وحدث هلع في الوظيفة، فإن مشغل الاختبار سيتعافي، بالرغم من أن الاختبار سيعتبر فاشلا بالطبع. إن الأخطاء المتوقعة مثل تلك الناتجة عن

مُدخل سيئ من المستخدم، أو الملفات المفقودة، أو التعديل غير السليم، يجب أن يقدم تقرير بها من خلال إعادة قيمة خطأ non-nil. لحسن الحظ مثال echo الذي قدمناه بسيط جدًا ولن يعيد خطأ non-nil أبدًا.

11.2.3 اختبار الصندوق الأبيض

إن أحد طرق تصنيف الاختبارات، هي تصنيفها وفقًا لمستوى المعرفة الذي تحتاجه من العمل الداخلي للحزمة قيد الاختبار. لا يفترض اختبار "الصندوق الأسود" (Black box) أي شيء حول الحزمة إلا ما يكشفه API الخاص بها، ويحدده توثيقها، والأجزاء الداخلية للحزمة تُعَد مصمتة لا تُرى. وعلى النقيض، اختبار "الصندوق الأبيض" (white-box) يمتلك إمكانية وصول مميزة للوظائف الداخلية وبنيات البيانات الخاصين بالحزمة، ويمكنه تقديم ملاحظات، وإجراء تغييرات لا يمكن للعميل العادي القيام بها. كمثال، يمكن لاختبار الصندوق الأبيض التحقق من أن ثوابت أنواع بيانات الحزمة يتم الحفاظ عليها بعد كل عملة. (إن اسم "الصندوق الأبيض" هو الاسم التقليدي، ولكن "الصندوق الشفاف" (clear box) سيكون أدقا).

إن الطريقتين مكملتان لبعضهما، فاختبارات الصندوق الأسود عادة ما تكون أقوى، وتحتاج تحديثات أقل مع تطور البرنامج، وهي تساعد مؤلف الاختبار أيضًا على التعاطف مع عميل الحزمة، ويمكن أن تكشف عن عيوب في تصميم API. على النقيض، يمكن لاختبارات الصندوق الأبيض تقديم تغطية أكثر تفصيلاً للأجزاء الأكثر صعوبة في التطبيق.

لقد رأينا بالفعل أمثلة على النوعين. فـ TestIsPalindrome يستدعي فقط الوظيفة المُصدّرة IsPalindrome، وبالتالي يُعتبر اختبار صندوق أسود، بينما TestEcho يستدعي وظيفة echo، ويحدث المتغير العالمي out، وكلاهما غير مُصدّر، مما يجعله اختبار صندوق أبيض.

عدلنا – أثناء تطوير TestEcho – وظيفة echo بحيث تستخدم المتغير على مستوى الحزمة "out" عند كتابة مُخرجها، وبالتالي يمكن للاختبار أن يحل محل المُخرج القياسي من خلال التطبيق البديل الذي يسجل البيانات من أجل فحصها لاحقًا. وباستخدام نفس التقنية، يمكننا استبدال أجزاء أخرى من شفرة الإنتاج من خلال التطبيقات "الزائفة" سهلة الاختبار. إن ميزة التطبيقات الزائفة هي أنها يمكن أن تكون أبسط في التعديل، وأكثر قابلية للتوقع، وموثوقة أكثر، ويمكن رصدها بسهولة أكثر. يمكن أن تتجنب بسهولة كذلك الآثار الجانبية غير المرغوب فيها، مثل تحديث قاعدة بيانات إنتاج أو شحن بطاقة ائتمان.

توضح الشفرة أدناه منطق فحص الباقة في خدمة ويب تقدم إمكانية التخزين الشبكي للمستخدمين. عندما يتجاوز المستخدمين 90% من باقتهم، يرسل لهم النظام رسالة بريد إلكتروني للتنبيه.

```
gopl.io/ch11/storage1
```

```
package storage
import (
    "fmt"
    "log"
    "net/smtp"
)
var usage = make(map[string]int64)
func bytesInUse(username string) int64 { return usage[username] }
// Email sender configuration.
// NOTE: never put passwords in source code!
const sender = "notifications@example.com"
const password = "correcthorsebatterystaple"
const hostname = "smtp.example.com"
const template = `Warning: you are using %d bytes of storage,
%d%% of your quota.`
func CheckQuota(username string) {
    used := bytesInUse(username)
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
    if percent < 90 {
        return // OK
    }
    msg := fmt.Sprintf(template, used, percent)
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
        []string{username}, []byte(msg))
    if err != nil {
        log.Printf("smtp.SendMail(%s) failed: %s", username, err)
    }
}
```

سنرغب في اختباره، ولكننا لا نريد اختباره بإرسال رسالة بريد إلكتروني حقيقية، وبالتالي ننقل منطق البريد الإلكتروني إلى وظيفة خاصة به، ونخزن تلك الوظيفة في متغير غير مُصدّر على مستوى الحزمة هو notifyUser:

```
gopl.io/ch11/storage2

var notifyUser = func(username, msg string) {
    auth := smtp.PlainAuth("", sender, password, hostname)
    err := smtp.SendMail(hostname+":587", auth, sender,
        []string{username}, []byte(msg))
    if err != nil {
        log.Printf("smtp.SendMail(%s) failed: %s", username, err)
    }
}
func CheckQuota(username string) {
    used := bytesInUse(username)
    const quota = 1000000000 // 1GB
    percent := 100 * used / quota
    if percent < 90 {
        return // OK
    }
    msg := fmt.Sprintf(template, used, percent)
    notifyUser(username, msg)
}
```

يمكننا الآن كتابة اختبار يستخدم آلية الإخطار الزائف البسيطة بدلاً من إرسال رسالة بريد إلكتروني حقيقية. يسجّل هذا الاختبار المستخدم المُخطَر ومحتويات الرسالة.

```
package storage
import (
    "strings"
    "testing"
)
func TestCheckQuotaNotifiesUser(t *testing.T) {
    var  notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ...simulate a 980MB-used condition...
    const user = "joe@example.org"
    CheckQuota(user)
    if notifiedUser == "" && notifiedMsg == "" {
        t.Fatalf("notifyUser not called")
    }
    if notifiedUser != user {
        t.Errorf("wrong user (%s) notified, want %s",
            notifiedUser, user)
    }
    const wantSubstring = "98% of your quota"
    if !strings.Contains(notifiedMsg, wantSubstring) {
        t.Errorf("unexpected notification message <<%s>>, "+
            "want substring %q", notifiedMsg, wantSubstring)
    }
}
```

هناك مشكلة واحدة مع ذلك: بعد عودة وظيفة الاختبار هذه، لن تعمل CheckQuota بعد الآن كما كانت لأنها لا زالت تستخدم تطبيق الاختبار الزائف notifyUsers. (هناك دائمًا خطر من هذا النوع عند تحديث المتغيرات العالمية). يجب أن نعدل الاختبار ونسترجع القيمة السابقة بحيث لا تقع أي آثار على الاختبارات التالية، ويجب أن نفعل كل هذا في مسارات التنفيذ، بما في ذلك حالات إخفاق وهلع الاختبار. يقترح هذا بالتالي استخدام defer.

```
func TestCheckQuotaNotifiesUser(t *testing.T) {
    // Save and restore original notifyUser.
    saved := notifyUser
    defer func() { notifyUser = saved }()
    // Install the test's fake notifyUser.
    var notifiedUser, notifiedMsg string
    notifyUser = func(user, msg string) {
        notifiedUser, notifiedMsg = user, msg
    }
    // ...rest of test...
}
```

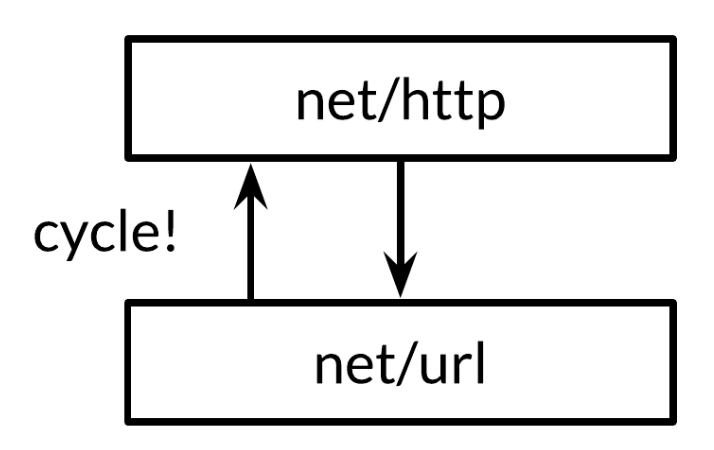
يمكن استخدام هذا النمط لحفظ واسترجاع كل أنواع المتغيرات العالمية مؤقتًا، وهذا يتضمن أعلام سطر الأوامر، وخيارات إصلاح الخلل، ومؤشرات الأداء، ولتثبيت وحذف مواضع الإضافة (hooks) التي تجعل شفرة الإنتاج تستدعي

شفرة اختبار ما عند حدوث شيء مثير للاهتمام، ولإدخال شفرة الإنتاج في حالات نادرة ولكن مهمة، مثل انتهاء المهلة، والأخطاء، وحتى التداخلات المحددة للأنشطة المتزامنة.

إن استخدام المتغيرات العالمية بهذه الطريقة آمن فقط لأن go test لا يُجرى اختبارات متعددة بالتزامن عادة.

11.2.4 حِزم الاختبارات الخارجية

انظر الحزمة net/url والتي تقدم مفسر URL، و net/http التي تقدم خادم ويب ومكتبة عميل HTTP. وكما يمكن أن نتوقع، فإن net/http ذات المستوى الأعلى، تعتمد على net/url ذات المستوى الأقل. مع ذلك، فإن أحد الاختبارات في net/url هو مثال يوضح التفاعل بين URLs ومكتبة عمل HTTP. بصيغة أخرى، اختبار الحزمة منخفضة المستوى يستورد حزمة عالية المستوى.

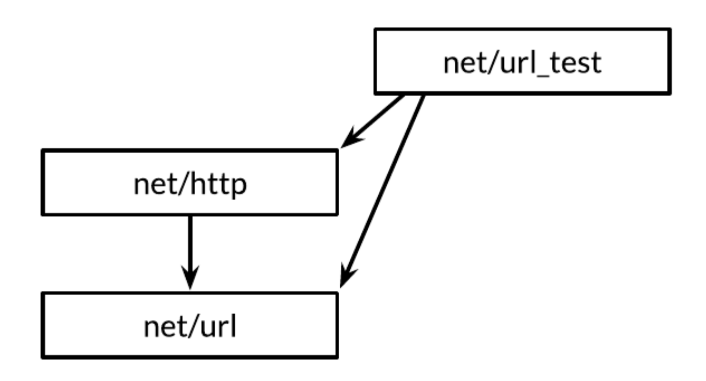


الشكل 11.1: اختبار net/url يعتمد على net/http.

إن إعلان وظيفة الاختبار هذه في حزمة net/url سيخلق دورة في مخطط استيراد الحزمة البياني، والتي يصورها السهم المتجه لأعلى في الشكل 11.1، ولكن كما شرحنا في القسم 10.1، فإن مواصفات Go تمنع دورات الاستيراد.

نحل هذه المشكلات من خلال إعلان وظيفة الاختبار في "حزمة اختبار خارجية" (external test package)، بمعنى..أننا نعلنها في ملف في دليل net/url، والذي يقرأ إعلان حزمته url_test الحزمة. إن اللاحقة الإضافية _test هي إشارة لـ go test تعني أنه يجب أن يبني حزمة إضافية تحتوي على هذه الملفات فقط وأن يجري الاختبارات عليها. قد يكون من المفيد التفكير في هذه الحزمة الخارجية كما لو أنها تحتوي على مسار الاستيراد net/url_test، ولكن لا يمكن استيرادها تحت هذا الاسم أو أي اسم آخر.

نظرًا لكون الاختبارات الخارجية تعيش في حزمة منفصلة، يمكنها استيراد حزم المساعِدة التي تعتمد كذلك على الحزمة التي تختبر، ولا يمكن للاختبار بداخل الحزمة فعل هذا. أما من حيث طبقات التصميم، فإن حزمة الاختبار الخارجية أعلى منطقيًا من كلتا الحزمتين اللتين تعتمد عليهما، كما هو موضح في الشكل 11.2.



الشكل 11.2: حزم اختبار خارجية تكسر دورات الاعتمادية.

إن تجنب دورات الاستيراد يسمح لحزم الاختبار الخارجية، وخاصة "اختبارات التكامل" (integration tests) (التي تختبر تفاعل مكونات متعددة) باستيراد حزم أخرى بحرية، كما سيفعل التطبيق بالضبط.

يمكننا استخدام أداة go list لتلخيص أي من ملفات مصدر Go في دليل الحزمة هي شفرة إنتاج، وأيها اختبارات داخل الحزمة، والاختبارات الخارجية. سنستخدم حزمة fmt كمثال. إن GoFiles هي قائمة بالملفات التي تحتوي على شفرة الإنتاج، وهي الملفات التي سيُدرجها go build في تطبيقك:

```
$ go list -f={{.GoFiles}} fmt
[doc.go format.go print.go scan.go]
```

إن TestGoFiles هو قائمة بالملفات التي تنتمي إلى حزمة fmt أيضًا، ولكن هذه الملفات – التي تنتهي أسماؤها كلها بـ _test.go – تُدرج فقط عند بناء الاختبارات:

```
$ go list -f={{.TestGoFiles}} fmt
[export_test.go]
```

توجد اختبارات الحزمة عادة في هذه الملفات، بالرغم من أن fmt لا تحتوي على أي منها عادة، وسنشرح هدف export_test.go في الجزء التالي.

إن XTestGoFiles هو قائمة الملفات التي تُشكل حزمة الاختبار الخارجية، fmt_test، وبالتالي يجب أن تستورد هذه الملفات حزمة fmt كي تتمكن من استخدامها. مرة أخرى، تُدرج هذه الملفات فقط خلال الاختبار:

```
$ go list -f={{.XTestGoFiles}} fmt
[fmt_test.go scan_test.go stringer_test.go]
```

قد تحتاج حزمة الاختبار الخارجية أحيانًا إلى إمكانية دخول مميزة إلى الأجزاء الداخلية من الحزمة قيد الاختبار، لو كان اختبار صندوق أبيض مثلاً يجب أن يتم في حزمة منفصلة لتجنب دورة الاستيراد. نستخدم في هذه الحالات الخدعة التالية: نضيف إعلانات إلى ملف الحزمة الداخلية test.go_ لكشف البيانات الداخلية الضرورية أمام الاختبار الخارجي. من ثمّ، يقدم هذا الملف للاختبار "باب خلفي" للحزمة. لو كان ملف المصدر موجودا فقط لهذا الهدف ولا يحتوي على ملفات اختبار، فإنه يُطلق عليه عادة export_test.go.

على سبيل المثال، تطبيق حزمة fmt يحتاج إلى أداء unicode.IsSpace كجزء من fmt.Scanf، ولتجنب خلق اعتمادية غير مرغوب فيها، لا تستورد fmt حزمة Unicode وجداول بياناتها الكبيرة، بل بدلاً من ذلك، تحتوي على تطبيق أبسط، تُطلق عليه isSpace.

لضمان أن سلوكيات fmt.isSpace و unicode.IsSpace لا تنحرف بعيدًا عن بعضها، تحتوي fmt على اختبار. وهو اختبار خارجي، وبالتالي لا يمكنه الدخول لـ isSpace مباشرة، وبالتالي تفتح fmt بابًا خلفيًا له من خلال إعلان متغير مُصدّر يحمل وظيفة isSpace الداخلية. إن هذا هو كامل ملف export_test.go في حزمة fmt.

```
package fmt
var IsSpace = isSpace
```

لا يُعرّف ملف الاختبار هذا أي اختبارات، بل يعلن عن الرمز المُصدر fmt.IsSpace فقط ليستخدمه الاختبار الخارجي. يمكن استخدام هذه الخدعة كذلك كلما احتاج الاختبار الخارجي لاستخدام بعض تقنيات اختبار الصندوق الأبيض.

11.2.5 كتابة اختبارات فعالة

يتفاجأ معظم من يتعاملون لأول مرة مع Go بمدى بساطة إطار اختبار Go. تقدم إطارات اللغات الأخرى آليات لتحديد وظائف الاختبار (تستخدم عادة الانعكاس أو البيانات الوصفية)، وإضافات للروتين (hooks) لأداء عمليات "setup" و "teardown" قبل وبعد الاختبار، ومكتبات لوظائف الوسائل لتأكيد الفرضيات الأساسية، ومقارنة القيم، وتهيئة رسائل الخطأ، وإجهاض الاختبار الفاشل (باستخدام الاستثناءات عادة). بالرغم من أن هذه الآليات يمكن أن تجعل الاختبارات دقيقة جدًا، إلا أن الاختبارات الناتجة عادة ما تبدو وكأنها مكتوبة بلغة أجنبية. علاوة على ذلك، وبالرغم من أن بإمكانهم ذِكر حدوث PASS أو FAIL بطريقة صحيحة، إلا أن طريقتهم قد لا تكون جيدة في التعامل مع المصين (maintainer) سيء الحظ، نتيجة رسائل الفشل الغامضة مثل "assert: 0 == 1" أو صفحة وراء صفحة من تتبعات الرصة.

إن موقف Go من الاختبار هو النقيض من هذا تمامًا، فهو يتوقع من مؤلفي الاختبار أن يقوموا بمعظم عملهم بأنفسهم، وتحديد وظائف لتجنب التكرار، كما يفعلون في البرامج العادية. إن عملية الاختبار ليست مجرد ملئ روتيني لاستمارات، بل إن الاختبار له واجهة مستخدم أيضًا، وإن كان الفارق هو أن مستخدميه الوحيدين هم المصينين (maintainers) له

أيضًا. إن الاختبار الجيد لا ينفجر في حالة الفشل ولكنه يطبع وصفا واضحا ودقيقا لأعراض المشكلة، وربما بعض الحقائق ذات الصلة حول سياقها. في الوضع المثالي، لا يجب أن يحتاج المصين لقراءة شفرة المصدر لفك شفرة فشل الاختبار، والاختبار الجيد لا يجب أن يستسلم بعد حالة فشل واحدة، ولكن يجب أن يحاول تقديم تقرير بأخطاء متعددة في التشغيل الواحد، حيث أن نمط الإخفاقات قد يكون كاشفا لسببها.

إن وظيفة التأكيد أدناه تقارن بين قيمتين، وتبني رسالة خطأ عامة، وتوقف البرنامج. إنها دقيقة وسهلة الاستخدام، ولكن عندما تفشل فإن رسالة الخطأ تكاد تكون عديمة الفائدة. وهي لا تحل المشكلة الصعبة الخاصة بتقديم واجهة مستخدم جيدة.

```
import (
   "fmt"
   "strings"
   "testing"
)
// A poor assertion function. func
assertEqual(x, y int) { if x != y {
panic(fmt.Sprintf("%d != %d", x, y))
}
}
func TestSplit(t *testing.T) {
   words := strings.Split("a:b:c", ":")
   assertEqual(len(words), 3)
   // ...
}
```

بهذا المنطق، تعاني وظائف التأكيد من "التجريد السابق لأوانه" (premature abstraction)، ومن خلال معاملة فشل هذا الاختبار المحدد كمجرد اختلاف بين عددين صحيحين، سنتمكن من استغلال فرصة تقديم سياق ذو معنى. يمكننا تقديم رسالة أفضل من خلال البدء من التفاصيل الملموسة، كما هو الحال في المثال أدناه. وبمجرد ظهور أنماط تكراره في مجموعة اختبار يمكننا تقديم التجريدات.

```
func TestSplit(t *testing.T) {
   s, sep := "a:b:c",
   words : strings.Split(s, sep)
   if got, want := len(words), 3; got != want {
      t.Errorf("Split(%q, %q) returned %d words, want %d",
         s, sep, got, want)
   }
   // ...
}
```

يقدم الاختبار الآن الوظيفة التي استدعيناها، ومُدخلاتها، وأهمية نتيجتها، وهو يحدد صراحة القيمة الفعلية والتوقعات، وهو يواصل التنفيذ حتى لو فشل هذا التأكيد. بعد أن نكتب اختبار كهذا، ستكون الخطوة الطبيعية التالية هي عدم

تعريف وظيفة لاستبدال عبارة if بأكملها، وإنما تنفيذ الاختبار في حلقة يتفاوت فيها كل من s, sep و want، مثل الاختبار المدفوع بالجدول في IsPalindrome.

لم يحتَج المثال السابق لأي وظائف وسيلة، ولكن هذا لا يجب أن يوقفنا بالتأكيد عن تقديم الوظائف لو كانت ستساعد على جعل الشفرة أبسط. (سنلقي نظرة على واحدة من وظائف الوسيلة هذه، reflect.DeepEqual، في القسم 13.3). إن مفتاح الاختبار الجيد هو البدء بتطبيق السلوك الملموس الذي تريده، واستخدام الوظائف حينها فقط لتبسيط الشفرة وحذف التكرار. نادرًا ما يتم الحصول على أفضل نتائج لو بدأت بمكتبة وظائف الاختبار المجردة العامة.

تمرين 11.5: قم بتوسيع TestSplit ليشمل استخدام جدول المدخلات والمُخرجات المتوقعة.

11.2.6 تجنب اختبارات بريتل - Brittle Tests

إن التطبيق الذي يحدث به إخفاق عادة عند مواجهة مدخلات جديدة وصحيحة يُطلق عليه buggy، والاختبار الذي يفشل بشكل لافت للنظر عند إجراء تغيير سليم على البرنامج يُطلق عليه brittle. ومثلما يُحبط برنامج buggy مستخدمة، فإن اختبار brittle يثير سخط مصينيه. إن معظم اختبارات brittle – التي تفشل في أي تغيير تقريبًا في شفرة الإنتاج سواء جيد أو سيء – يُطلق عليها أحيانًا اختبارات "كاشف التغيير" (change detector) أو "الوضع الراهن" (status quo)، والوقت المقضي في التعامل معهم يمكن أن يستنزف سريعًا أي منافع قدموها في وقت ما.

عندما تنتج وظيفة قيد الاختبار نتيجة معقدة مثل سلسلة طويلة، وبنية بيانات مفصلة، أو ملف، سيكون من المغري التحقق مما إذا كان الناتج مساوي بالضبط للقيمة "الذهبية" التي كانت متوقعة عند كتابة الاختبار. ولكن مع تطور البرنامج، من المرجح أن تتغير أجزاء من الناتج كذلك، غالبا سيكون تغيير جيدا ولكنه يظل تغييرًا رغم كل شيء. ليس الأمر متعلق بالناتج فقط، بل أن الوظائف في المُدخلات المعقدة كذلك عادة ما تتفكك لأن المُدخل المستخدم في اختبار لم يعد صحيحًا.

إن أسهل طريقة لتجنب اختبارات brittle هي التحقق من الروابط المناسبة التي تهتم بها، اختبر واجهات برنامجك الأبسط والأكثر استقرارًا بشكل يفضّل وظائفه الداخلية. كن انتقائيًا في تأكيداتك، ولا تتحقق من المطابقة مع السلسلة بالضبط، ولكن ابحث عن السلاسل الفرعية التي ستظل دون تغيير مع تطور البرنامج. يستحق الأمر عادة كتابة وظيفة جوهرية لتقطير الناتج المعقد إلى خلاصته بحيث تكون التأكيدات موثوقة. بالرغم من أن هذا قد يبدو جهدًا زائدًا إلا أنه سيحقق فوائد سريعًة بدلاً من إضاعة الوقت في إصلاح اختبارات فاشلة.

11.3 التغطية - Coverage

إن الاختبار لا يكتمل أبدًا بطبيعته، أو كما يقول عالِم الكمبيوتر المؤثّر " Edsger Dijkstra": "يوضح الاختبار وجود الخلل وليس غيابه". لا يوجد كم كافي من الاختبار يمكن أن يثبت أن الحزمة خالية من الاختلالات. في أفضل الأحوال، ستزيد ثقتها وحسب في أن الحزمة تعمل جيدًا في نطاق واسع من السيناريوهات المهمة.

إن درجة تفعيل الاختبار للحزمة قيد الاختبار يُطلق عليه "تغطية الاختبار" (test's coverage). لا يمكن تحديد التغطية كميًا بشكل مباشر – فديناميكيات كل البرامج باستثناء البرامج شديدة التفاهة تتجاوز قدرتنا على القياس الدقيق – ولكن هناك استدلالات يمكنها مساعدتنا على توجيه اختبارات جهودنا حيث يُرجح أن تحقق أكبر فائدة ممكنة.

إن تغطية العبارة (Statement coverage) هي أبسط استدلال وأكثر الاستدلالات شيوعًا. إن تغطية العبارة الخاصة بمجموعة الاختبار هي جزء من عبارات المصدر التي تُنفذ مرة واحدة على الأقل خلال الاختبار. وسنستخدم في هذا القسم أداة جو cover ، والتي تُدمج في go test لقياس تغطية العبارة، والمساعدة على تحديد الفجوات الواضحة في الاختبارات.

إن الشفرة أدناه هي اختبار مدفوع بالجدول وهي خاصة بمُقيّم التعبير (expression evaluator) الذي بنيناه في الفصل السابع.

```
gopl.io/ch7/eval
func TestCoverage(t *testing.T) {
    var tests = []struct {
        input string
        env   Env
        want  string // expected error from Parse/Check or result from Eval
    }{
        {"x % 2", nil, "unexpected '%'"},
        {"!true", nil, "unexpected '!'"},
        {"log(10)", nil, `unknown function "log"`},
        {"sqrt(1, 2)", nil, "call to sqrt has 2 args, want 1"},
        {"sqrt(A / pi)", Env{"A": 87616, "pi": math.Pi}, "167"},
        {"pow(x, 3) + pow(y, 3)", Env{"x": 9, "y": 10}, "1729"},
        {"5 / 9 * (F - 32)", Env{"F": -40}, "-40"},
    }
    for _, test := range tests {
        expr, err := Parse(test.input)
        if err == nil {
            err = expr.Check(map[Var]bool{})
        }
        if err != nil {
            if err.Error() != test.want {
                t.Errorf("%s: got %q, want %q", test.input, err, test.want)
            }
            continue
        }
        got := fmt.Sprintf("%.6g", expr.Eval(test.env))
        if got != test.want {
            t.Errorf("%s: %v => %s, want %s",
                test.input, test.env, got, test.want)
        }
    }
```

```
}
```

لنتأكد أولاً أن الاختبار نجح:

```
$ go test -v -run=Coverage gopl.io/ch7/eval
=== RUN TestCoverage
--- PASS: TestCoverage (0.00s)
PASS
ok  gopl.io/ch7/eval 0.011s
```

يُظهر هذا الأمر رسالة الاستخدام في أداة التغطية:

```
$ go tool cover
Usage of 'go tool cover':
Given a coverage profile produced by 'go test':
 go test -coverprofile=c.out
Open a web browser displaying annotated source code:
go tool cover -html=c.out
...
```

يشغل أمر go tool واحدا من الملفات التنفيذية من سلسلة أدوات Go. تسكن هذه البرامج في الدليل:

$GOROOT/pkg/tool/${GOOS}_${GOARCH}، وبفضل go build، نادرًا ما نحتاج لتفعيلها بشكل مباشر.

سنجري الاختبار الآن باستخدام عَلَم coverprofile– :

```
$ go test -run=Coverage -coverprofile=c.out gopl.io/ch7/eval
ok  gopl.io/ch7/eval 0.032s coverage: 68.5% of statements
```

يسمح هذا العَلَم بجمع بيانات التغطية من خلال "توجيه" (instrumenting) شفرة الإنتاج. بمعنى، أنه يعدّل نسخة من شفرة المصدر بحيث يتم ضبط متغير منطقي (Boolean) قبل تنفيذ كل بنية من بنيات العبارات، وتحديد متغير واحد لكل بِنية. وقبل أن يخرج البرنامج المعدل، يكتب قيمة لكل متغير في ملف السجل المحدد c.out، ويطبع ملخص بجزء من العبارات التي نُفذت. (لو كان كل ما تحتاجه هو الملخص، استخدم go test -cover).

لو شغل go test مع عَلَم covermode=count- ، فإن توجيه كل بِنية يزيد العداد بدلاً من تعيين متغير منطقي (Boolean). إن سجل عداد التنفيذ الناتج الخاص بكل بنية يسمح بالمقارنات الكمية بين البنيات "الأسخن"، والتي تُنفذ بتكرار أكبر، وبين البنيات "الأبرد".

بعد أن جمعنا البيانات، سنشغل أداة التغطية (cover tool)، والتي تعالج السجل، وتنتج تقرير HTML، وتفتحه في نافذة متصفح جديد (الشكل 11.3).

```
$ go tool cover -html=c.out
```

```
coverage.html
file:///home/gopher/gobook/coverage.html
gopl.io/ch7/eval/eval.go (58.8%)   not tracked   not covered   covered

func (u unary) Eval(env Env) float64 {
        switch u.op {
        case '+':
                return +u.x.Eval(env)
        case '-':
                return -u.x.Eval(env)
        }
        panic(fmt.Sprintf("unsupported unary operator: %q", u.op))
}

func (b binary) Eval(env Env) float64 {
        switch b.op {
        case '+':
                return b.x.Eval(env) + b.y.Eval(env)
        case '-':
                return b.x.Eval(env) - b.y.Eval(env)
        case '*':
                return b.x.Eval(env) * b.y.Eval(env)
        case '/':
                return b.x.Eval(env) / b.y.Eval(env)
        }
        panic(fmt.Sprintf("unsupported binary operator: %q", b.op))
}
```

الشكل 13: تقرير تغطية.

تُلوّن كل عبارة باللون الأخضر لو كانت مغطاة، وباللون الأحمر لو لم تكن مُغطاة. وللتوضيح، ظللنا خلفية النص المكتوب باللون الأحمر. ويمكننا أن نرى فورًا أن أي من مدخلاتنا لم يمارس طريقة Eval الخاصة بالمشغل الأحادي. لو أضفنا حالة الاختبار الجديدة هذه إلى الجدول، وأعيد إجراء الأمرين السابقين، فإن شفرة التعبير الأحادي تصبح ذات لون أخضر:

```
{"-x * -x", eval.Env{"x": 2}, "4"}
```

ستظل عبارتيّ الهلع باللون الأحمر مع ذلك. لا يجب أن يكون هذا مفاجئا، لأن هذه العبارات يُفترض ألا يوصل لها.

إن تحقيق تغطية 100٪ للعبارة يبدو هدفا نبيلا، ولكن ليس ذو جدوى عادة في الممارسة العملية، وليس استغلال جيد للجهود في أرجح الأحوال. وفقط لأن العبارة نُفذت فإن هذا لا يعني أنه خالية من الأخطاء، فالعبارات التي تحتوي على تعبيرات معقدة يجب تنفيذها مرات متعددة بمدخلات مختلفة لتغطية الحالات المثيرة للاهتمام. هناك بعض العبارات،

مثل عبارات الهلع المذكورة أعلاه، لا يمكن الوصول لها أبدًا. بينما هناك بعض العبارات الأخرى، مثل تلك الخاصة بالتعامل مع الأخطاء المقصورة على فئة معينة، من الصعب ممارستها ولكن نادرًا ما يتم الوصول لها في الواقع العملي. إن الاختبار هو مسعى نفعي بشكل أساسي، وهو مقايضة بين تكلفة كتابة الاختبارات وتكلفة الإخفاقات التي كان يمكن أن تمنعها الاختبارات. يمكن لأدوات التغطية المساعدة في تحديد أضعف النقاط، ولكن ابتكار حالات اختبار جيدة تحتاج لنفس التفكير الصارم مثلها مثل البرمجة بشكل عام.

11.4 وظائف قياس الأداء

إن قياس الأداء هو ممارسة قياس أداء البرنامج وفقًا لِحمل عمل ثابت. وفي لغة go، تبدو وظيفة قياس الأداء كوظيفة اختبار، ولكن مع سابقة "Benchmark" ومعامل testing.B* الذي يقدم تقريبًا نفس الطُرق التي يقدمها testing.T*، إضافة إلى بعض الأشياء القليلة الزائدة المتعلقة بقياس الأداء. وهو يكشف أيضًا عن حقل العدد الصحيح N، والذي يحدد عدد مرات أداء العملية التي يتم قياسها.

إليك قياس أداء لـ IsPalindrome يستدعيها عدد N من المرات في حلقة:

```
import "testing"
func BenchmarkIsPalindrome(b *testing.B) {
    for i := 0; i < b.N; i++ {
        IsPalindrome("A man, a plan, a canal: Panama")
    }
}
```

نحن نشغلها باستخدام الأمر أدناه، وعلى العكس من الاختبارات، فإن الوضع الاعتيادي لا ينفذ أي قياسات أداء. إن المعطى لعلَم bench- يختار قياسات الأداء التي يُجريها، وهو تعبير منتظم يطابق أسماء وظائف Benchmark، مع قيمة اعتيادية لا تطابق أي منهم. إن نمط "." يجعله يطابق كل قياسات الأداء في حزمة word، ولكن حيث أنه لا يوجد سوى واحدة فقط، فإن bench=IsPalindrome- سيكون مكافئ لها.

```
$ cd $GOPATH/src/gopl.io/ch11/word2
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000 1035 ns/op
ok   gopl.io/ch11/word2     2.179s
```

إن اللاحقة الرقمية لاسم أداء القياس، وهي 8 هنا، تشير إلى قيمة GOMAXPROCS، وهي مهمة لقياسات الأداء المتزامنة.

يخبرنا التقرير بأن كل استدعاء لـ IsPalindrome يستغرق حوالي 1.035 ميكروثانية، بمتوسط 1,000,000تشغيل. وحيث أن مشغل قياس الأداء ليس لديه أي فكرة مبدئيًا عن طول الفترة التي تستغرقها العملية، فإنه يقدم بعض القياسات المبدئية باستخدام قيم N صغيرة، ثم يستقرأ إلى قيمة كبيرة بما يكفي لإجراء قياس مستقر للتوقيت.

إن سبب تطبيق الحلقة بواسطة وظيفة أداء قياس وليس بواسطة شفرة استدعاء في تعريف الاختبار هو أن وظيفة الأداء لديها فرصة تنفيذ شفرة الإعداد الضرورية لمرة واحدة خارج الحلقة بدون أن يضيف هذا شيئًا إلى الوقت المُقاس الخاص بكل تكرار. لو كانت شفرة الإعداد هذه لا زالت تسبب اضطراب في النتائج، فإن مؤشر testing.B سيقدم طرق لإيقاف أو متابعة أو إعادة تعيين المؤقّت، ولكن نادرًا ما يكون هناك حاجة لهذا.

بعد أن أصبح لدينا الآن قياسات أداء واختبارات، أصبح من السهل أن نجرب خارجيًا كيف نجعل البرنامج أسرع. ربما يكون التحسين الواضح هو جعل حلقة IsPalindrome الثانية تتوقف عن التحقق في المنتصف، لتجنب القيام بكل مقارنة مرتين:

```
n := len(letters)/2
for i := 0; i < n; i++ {
    if letters[i] != letters[len(letters)-1-i] {
        return false
    }
}
return true
```

ولكن كما هو الحال عادة، فإن التحسين الواضح لا ينتج عنه الفائدة المتوقعة دائمًا. وحقق هذا تحسين 4% فقط في تجربة واحدة.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 1000000 992 ns/op
ok  gopl.io/ch11/word2    2.093s
```

إن أحد الأفكار الأخرى هي التخصيص المسبق لمصفوفة كبيرة بما يكفي لاستخدامها بواسطة letters، بدلاً من التوسع بواسطة استدعاءات متتالية لـ append. إن إعلان أن letter هي مصفوفة بحجم مناسب، هكذا:

```
letters := make([]rune, 0, len(s))
for r := range s {
    if unicode.IsLetter(r) {
        letters = append(letters, unicode.ToLower(r))
    }
}
```

ينتج عنه تحسن يقترب من 35%، ويقدم مشغل أداة القياس الآن متوسط يزيد عن 2,000,000 تكرار.

```
$ go test -bench=.
PASS
BenchmarkIsPalindrome-8 2000000 697 ns/op
ok  gopl.io/ch11/word2     1.468s
```

كما يوضح هذا المثال، فإن أسرع برنامج عادة ما يكون البرنامج الذي يقدم أقل تخصيصات للذاكرة. وعلم سطر الأوامر benchmem- سيتضمن إحصائيات تخصيص الذاكرة في تقريره. نحن نقارن هنا عدد من التخصيصات قبل التحسين:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome 1000000   1026 ns/op 304 B/op 4 allocs/op
```

وبعدها:

```
$ go test -bench=. -benchmem
PASS
BenchmarkIsPalindrome 2000000   807 ns/op 128 B/op 1 allocs/op
```

إن دمج التخصيصات في استدعاء واحد لـ make يمسح 75% من التخصيصات، ويقسم نصف كمية الذاكرة المخصصة للنصف.

إن قياسات الأداء هذه تخبرنا بالوقت النهائي المطلوب لعملية معينة، ولكن تدور الأسئلة المثيرة للاهتمام حول الأداء في العديد من المواقف حول التوقيتات النسبية لعمليتين مختلفتين. كمثال، لو كانت وظيفة تستغرق 1مث لمعالجة 1000 عنصر، فكم ستستغرق معالجة 10,000 أو مليون؟ تكشف هذه المقارنات على النمو المتقلب في وقت تشغيل الوظيفة. المثال الآخر هو: ما هو أفضل حجم لصوان I/O؟ إن قياسات أداء إنتاجية التطبيق في نطاق أحجام متنوعة يمكن أن يساعدنا على اختيار أقل صوان يقدم أداء مُرضي. المثال الثالث: ما هي الخوارزمية التي تؤدي أفضل أداء في وظيفة معينة؟ إن قياسات الأداء التي تقيم خوارزميتين مختلفتين في نفس بيانات المُدخل يمكن أن تُظهر عادة نقاط قوة وضعف في كل واحدة منهما في أحمال العمل المهمة أو التمثيلية.

إن قياسات الأداء المقارنة هي مجرد شفرة منتظمة، وهي تأخذ عادة شكل وظيفة بارامترية منفردة تُستدعى من وظائف Benchmark المتعددة ذات القيم المختلفة، كالمثال التالي:

```
func benchmark(b *testing.B, size int) { /* ... */ }
func Benchmark10(b *testing.B) { benchmark(b, 10) }
func Benchmark100(b *testing.B) { benchmark(b, 100) }
func Benchmark1000(b *testing.B) { benchmark(b, 1000) }
```

إن معامل size ، الذي يحدد حجم المُدخل، يتفاوت عبر قياسات الأداء، ولكنه ثابت داخل كل قياس أداء. قاوم إغراء استخدام معامل b.N كحجم مُدخل، وما لم تفسره كتعداد تكرار لمُدخل ثابت الحجم، فإن نتائج قياس الأداء الخاص بك ستكون عديمة المعنى.

إن الأنماط التي كشفتها قياسات الأداء المقارنة مفيدة بشكل خاص خلال تصميم البرنامج، ولكننا لا نرمي قياسات الأداء عندما يعمل البرنامج. مع تطور البرنامج، أو مع نمو مُدخلاته، أو عند نشره على نظم تشغيل جديدة أو معالجات ذات خصائص مختلفة، يمكننا إعادة استخدام قياسات الأداء هذه للعودة إلى قرارات التصميم هذه.

تمرين 11.6: اكتب قياسات أداء لمقارنة تطبيق PopCount في القسم 2.6.2 مع حلولك للتمرين 2.4 والتمرين 2.5. في أي نقطة يصل المنهج المعتمد على الجدول إلى نقطة التعادل؟

تمرين 11.7: اكتب قياسات الأداء لـ Add و UnionWith والطُرق الأخرى في IntSet* (انظر 6.5) باستخدام مُدخلات كبيرة شِبه عشوائية. ما مستوى السرعة الذي يمكنك تشغيل هذه الطُرق بها؟ كيف يؤثر اختيار حجم الكلمة على الأداء؟ ما مدى سرعة IntSet مقارنة بتطبيق محدد معتمد على نوع خريطة مدمجة؟

11.5 التنميط - Profiling

إن قياسات الأداء مفيدة في قياس أداء عمليات محددة، ولكن عندما نحاول جعل برنامج بطيء يعمل بشكل أسرع، لا يكون لدينا أي فكرة عادة من أين نبدأ. يعرف كل مبرمج قول "دونالد نوث" (Donald Knuth) حول التحسين السابق لأوانه، والذي ظهر في "البرمجة الهيكلية في عبارات go to" عام 1974. بالرغم من أنه يُساء تفسيره عادة بحيث يعني أن الأداء غير مهم، إلا أن معناه في سياقه الأصلي يقدم معنى مختلفا تمامًا:

> "لا شك في أن ذروة الكفاءة تؤدي إلى سوء الاستخدام، ويضيع المبرمجون كما هائلا من الوقت في التفكير والقلق بشأن أجزاء غير حَرِجة من برامجهم، وهذه المحاولات لتحقيق الكفاءة سيكون لها تأثير سلبي قوي فعليًا عند محاولة اكتشاف الخلل وإجراء الصيانة. يجب أن ننسى العيوب الصغيرة، ففي حوالي 97% من الوقت تقريبًا يُعد التحسين السابق لأوانه السبب الجذري لكل المشكلات.
>
> مع ذلك، لا يجب أن نجازف بفرصنا في الـ 3% الباقية، والمبرمج الجيد لن يغريه الرضا الناتج عن هذا المنطق، وسيكون حكيما بما يكفي ليبحث بدقة في الشفرة الحَرِجة، ولكن بعد تحديد الشفرة. سيكون من الخطأ عادة إصدار أحكام مسبقة حول أجزاء البرنامج الحرجة حقًا، حيث أن الخبرة

العامة التي وصلها لها المبرمجون الذين يستخدمون أدوات القياس هي أن تخميناتهم المبدئية تفشل عادة."

عندما نرغب في بحث سرعة برامجنا بدقة، سنجد أن أفضل تقنية لتحديد الشفرة الحرجة هي "التنميط" (Profiling). إن التنميط هو طريقة آلية لقياس الأداء بناء على أخذ عينة من عدد من أحداث الملف خلال التنفيذ، ثم الاستقراء منهم أثناء خطوة ما بعد المعالجة، ويُطلق على الملخص الإحصائي الناتج "مجموعة الصفات العامة/النمط" (Profile).

تدعم Go أنواع تنميط متعددة، كل منها معنى بجانب أداء مختلف، ولكنها كلها تتضمن تسجيل تسلسل أحداث محل اهتمام، وكل منها له تتبع رصة مصاحب له – رصة استدعاءات الوظيفة النشطة في لحظة الحدث. إن أداة go test تمتلك دعما مدمجا لأنواع تنميط متعددة.

يحدد "نمط CPU" (CPU Profile) الوظائف التي يحتاج تنفيذها إلى معظم وقت الـ CPU، وتُقاطع الخيوط المشغلة حاليًا في كل CPU دوريًا بواسطة نظام التشغيل كل بضع مللي ثانية، وتسجل كل مقاطعة حدث نمط واحد قبل متابعة التنفيذ الاعتيادي.

يحدد "نمط التكدُس" (heap profile) العمليات المسؤولة عن حجز معظم الذاكرة. تقوم مكتبة التنميط بتجميع الاستدعاءات إلى حجز الذاكرة الداخلية الروتينية بمتوسط تسجيل كل حدث تنميط لكل 512KB من الذاكرة المحجوزة.

يحدد " نمط الحظر " (blocking profile) العمليات المسؤولة عن حظر ال goroutines لأطول فترة ممكنة، مثل استدعاءات النظام، ومرسلات واستلامات القناة، والاستحواذات على الأقفال. تسجل مكتبة التنميط حدث في كل مرة يُحظر goroutine بواسطة واحدة من هذه العمليات.

إن تجميع نمط أو صفات عامة أساسية لشفرة تحت الاختبار سهل مثله مثل إتاحة واحدة من الأعلام أدناه، ولكن انتبه عند استخدام أكثر من علم واحد في المرة الواحدة لأن آلية جمع نوع واحدة من الصفات أو الأنماط يمكن أن تؤدي لانحراف نتائج الأنماط الأخرى.

```
$ go test -cpuprofile=cpu.out
$ go test -blockprofile=block.out
$ go test -memprofile=mem.out
```

من السهل أيضا إضافة دعم التنميط إلى برامج أخرى غير الاختبارات، بالرغم من أن تفاصيل كيفية فعلنا لهذا تتفاوت بين أدوات سطر الأوامر قصيرة العمر وبين تطبيقات الخادم طويلة المدى. إن التنميط مفيد بشكل خاص في البرامج التي تعمل على المدى الطويل، وبالتالي فإن خصائص التنميط لزمن تشغيل Go يمكن تفعيلها عن طريق تحكم المبرمج باستخدام runtime API.

بمجرد أن نجمع الصفات الأساسية (النمط)، سنحتاج إلى تحليلها باستخدام أداة pprof، وهذا هو الجزء القياسي من توزيعة Go، ولكن حيث أنها ليست أداة يومية، فإنها تُدخل مباشرة باستخدام go tool pprof. وهي تحتوي على عشرات الخصائص والخيارات، ولكن الاستخدام الأساسي يتطلب معطيين فقط، وهما الملف التنفيذي الذي ينتج النمط، وسجل النمط.

لا يتضمن السجل أسماء الدوال وهذا لجعل التنميط أكثر كفاءة ولتوفير المساحة، بدلاً من ذلك، تُحدد الوظائف بواسطة عناوينها. معنى هذا أن pprof يحتاج إلى ملف تنفيذي من أجل فهم السجل، وبالرغم من أن go test عادة ما يتخلص من الملف التنفيذي للاختبار بمجرد اكتمال الاختبار، إلا أنه عند إتاحة التنميط، فإنه يحفظ الملف التنفيذي كـ foo.test، حيث foo هو اسم الحزمة المُختبرة.

توضح الأوامر أدناه كيفية جمع وعرض نمط CPU بسيط. ولقد اخترنا واحد من قياسات الأداء من حزمة net/http. من الأفضل عادة تنميط قياسات أداء محددة بُنيت لتمثيل أحمال العمل التي يهتم الفرد بها. إن حالات اختبار قياس الأداء لا تكون ممثلة أبدًا تقريبًا، وهذا هو السبب في كوننا نلغي تفعيلهم باستخدام المرشح -run=NONE .

```
$ go test -run=NONE -bench=ClientServerParallelTLS64 \
    -cpuprofile=cpu.log net/http
PASS
BenchmarkClientServerParallelTLS64-8  1000
    3141325 ns/op 143010 B/op 1747 allocs/op
ok  net/http 3.395s
$ go tool pprof -text -nodecount=10 ./http.test cpu.log
2570ms of 3590ms total (71.59%)
Dropped 129 nodes (cum <= 17.95ms)
Showing top 10 nodes out of 166 (cum >= 60ms)
     flat  flat% sum%  cum cum%
    1730ms  48.19% 48.19% 1750ms 48.75% crypto/elliptic.p256ReduceDegree
    230ms  6.41% 54.60%  250ms 6.96%  crypto/elliptic.p256Diff
    120ms  3.34% 57.94%  120ms 3.34%  math/big.addMulVVW
    110ms  3.06% 61.00%  110ms 3.06%  syscall.Syscall
     90ms  2.51% 63.51%  1130ms 31.48%  crypto/elliptic.p256Square
     70ms  1.95% 65.46%  120ms 3.34%  runtime.scanobject
     60ms  1.67% 67.13%  830ms 23.12%  crypto/elliptic.p256Mul
     60ms  1.67% 68.80%  190ms 5.29%  math/big.nat.montgomery
     50ms  1.39% 70.19%  50ms 1.39%  crypto/elliptic.p256ReduceCarry
     50ms  1.39% 71.59%  60ms 1.67%  crypto/elliptic.p256Sum
```

يحدد علم text- هيئة الناتج، وفي هذه الحالة، سيكون الناتج عبارة عن جدول نصي به صف واحد لكل وظيفة، والتي تُرتب بحيث تظهر "أسخن" الوظائف أولاً – وهي التي تستهلك معظم دورات CPU. إن علم nodecount=10- يقيد النتائج ب 10 صفوف فقط. قد تكون هذه الهيئة النصية كافي لتحديد السبب في حالة مشكلات الأداء الإجمالية.

يخبرنا هذا النمط أن التشفير ذو المنحنى البيضاوي مهم بالنسبة لأداء قياس أداء HTTP المحدد هذا. على النقيض، لو كان النمط يهيمن عليه وظائف تخصيص الذاكرة من حزمة runtime، فإن تقليل استهلاك الذاكرة قد يكون تحسينًا جديرًا بالاهتمام.

قد يكون من الأفضل أن تستخدم واحد من عروض pprof الرسومية في حالة المشكلات الدقيقة أكثر. يتطلب هذا GraphViz، والذي يمكن تحميله من www.graphviz.org. إن علم web– ينتج رسم بياني مباشر لوظائف البرنامج، مرتبة وفقًا لأرقام نمط CPU الخاص بها، وملونة لتوضيح أسخن الوظائف.

لم نتعرض إلا إلى الملامح السطحية فقط لأدوات تنميط لغة Go، ولاكتشاف المزيد، اقرأ مقال "تنميط برامج Go" على مدونة go.

11.6 وظائف المثال - Example Functions

إن النوع الثالث من الوظائف يُعامل معاملة خاصة من go test وهو وظيفة المثال، وهي الوظيفة التي يبدأ اسمها بـ Example. وهي لا تحتوي على مؤشرات ولا على نتائج، وإليك وظيفة مثال خاصة بـ IsPalindrome:

```
func ExampleIsPalindrome() {
    fmt.Println(IsPalindrome("A man, a plan, a canal: Panama"))
    fmt.Println(IsPalindrome("palindrome"))
    // Output:
    // true
    // false
}
```

تخدم وظائف المثال ثلاثة أهداف. الهدف الأساسي هو التوثيق: يمكن للمثال الجيد أن يصبح طريقة دقيقة وبديهية لتوصيل سلوك وظيفة المكتبة أكثر من الوصف السردي، خاصة عند استخدامه كتذكير أو مرجع سريع. يمكن أن يوضح المثال كذلك التفاعل بين أنواع ووظائف متعددة تنتمي لـ API واحد، مثل إعلان النوع أو الوظيفة، أو الحزمة ككل. وعلى العكس من الأمثلة في التعليقات، تعد وظائف المثال شفرة Go حقيقية، وتخضع لفحص زمن الترجمة، حتى لا تنسى مع تطور الشفرة.

بناء على اللاحقة الخاصة بوظيفة المثال، سنجد أن خادم التوثيق المعتمد على الويب godoc يربط وظائف المثال على الوظيفة أو الخدمة التي يقدم مثالاً عليها، وبالتالي فإن ExampleIsPalindrome سيظهر مع توثيق وظيفة IsPalindrome، ووظيفة المثال المسماة Example فقط، سترتبط مع حزمة word ككل.

إن الهدف الثاني هو أن الأمثلة هي اختبارات التنفيذية تنفذها got test، ولو احتوت وظيفة المثال على تعليق // Output: كما هو الحال في المثال أعلاه، فإن تعريف الاختبار سينفذ الوظيفة ويتأكد من أن ما طبعته وفقًا لناتجها القياسي مطابق للنص داخل التعليق.

إن الهدف الثالث للمثال هو التجربة الفعلية. يستخدم خادم godoc في golang.org ملعب Go لجعل المستخدم يعدّل ويشغل كل وظيفة مثال من داخل متصفح الويب، كما هو موضح في الشكل 11.4. هذه عادة هي أسهل طريقة لجس نبض وظيفة معينة أو خاصية لغوية.

func **Join**

```
func Join(a []string, sep string) string
```

Join concatenates the elements of a to create a single string. The separator string sep is placed between elements in the resulting string.

▾ Example

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	s := []string{"foo", "bar", "baz"}
	fmt.Println(strings.Join(s, ", "))
}
```

```
foo, bar, baz

Program exited.
```

Run   Format   Share

الشكل 14: مثال تفاعلي على strings.Join في godoc.

يبحث آخر فصلين في هذا الكتاب حزم reflect و unsafe، والتي لا يستخدمهما سوى القليل من مبرمجي Go بانتظام، ويحتاج عدد أقل منهم لاستخدامها حقًا. لو لم تكتب أي برامج Go جوهرية بعد، سيكون الآن وقت مناسب لبدء كتابتها.

12- الانعكاس – Reflection

تقدّم Go آلية لتحديث المتغيرات، وفحص قيمها في زمن التشغيل، واستدعاء طُرقها، وتطبيق العمليات الجوهرية لتمثيلها، كل هذا من دون معرفة أنواعها في وقت الترجمة (Compile time). يُطلق على هذه الآلية اسم الانعكاس (reflection). إن الانعكاس يجعلنا نعامل الأنواع نفسها كقيم من الدرجة الأولى.

سنستكشف في هذا الفصل خصائص الانعكاس في لغة Go لنرى كيف تزيد قدرة اللغة على التعبير، وكيف تُعَد محورية لتطبيق اثنين من API المهمين وهما: تهيئة السلسلة النصية بواسطة fmt، وترميز البروتوكول عن طريق حِزم مثل encoding/json و encoding/xml. إن الانعكاس مهم أيضًا لآلية القالب التي تقدمها حزم text/template و html/template التي رأيناها في القسم 4.6. مع ذلك، فإن تبرير الانعكاس منطقيًا صعب، وليس للاستخدام العادي، لذا بالرغم من أن هذه الحِزَم تُطبق باستخدام الانعكاس، إلا أنها لا تكشف الانعكاس في APIs الخاصة بها.

12.1 لماذا الانعكاس؟

نحتاج أحيانًا إلى كتابة وظيفة قادرة على التعامل بشكل متسق مع قيم الأنواع التي لا ترضيها واجهة مشتركة، أو ليس لها تمثيل معروف، أو ليست موجودة في وقت تصميمنا للوظيفة – أو كل هذا معًا.

إن المثال المألوف على هذا هو منطق التهيئة fmt.Fprintf، والذي يمكن أن يطبع قيمة عشوائية من أي نوع بشكل مفيد، حتى لو كانت قيمة حددها المستخدم. لنحاول إعداد وظيفة كهذه باستخدام ما نعرفه بالفعل. وللتبسيط، ستقبل وظيفتنا معطى واحدة، وستعيد النتيجة كسلسلة مثلما تفعل fmt.Sprint، لذا سنطلق عليها Sprint.

سنبدأ بتبديل النوع يختبر ما إذا كان المعطى يُعرف طريقة String، وسنستدعيها لو كانت كذلك. سنضيف بعد ذلك حالات تبديل تختبر النوع الديناميكي للقيمة مقارنة بكل نوع من الأنواع الأساسية - string, int, bool، إلخ – ثم نجري عملية تهيئة في كل حالة:

```
func Sprint(x interface{}) string {
    type stringer interface {
        String() string
    }
    switch x := x.(type) {
    case stringer:
        return x.String()
    case string:
```

```
        return x
    case int:
        return strconv.Itoa(x)
        // ...similar cases for int16, uint32, and so on...
    case bool:
        if x {
            return "true"
        }
        return "false"
    default:
        // array, chan, func, map, pointer, slice, struct
        return "???"
    }
}
```

لكن كيف نتعامل مع الأنواع الأخرى، مثل float64, map[string] [ ]string[ ]، وغيرها؟ يمكننا إضافة مزيد من الحالات، ولكن عدد هذه الأنواع لا نهائي. وماذا عن الأنواع المُسمّاة، مثل url.Values؟ حتى لو كان تبديل النوع يحتوي على حالة مخصصة للنوع الضمني map[string][]string الخاص به، فإنها لن تتوافق مع url.Values لأن النوعين غير متطابقين، وتبديل النوع لا يمكن أن يتضمن حالة لكل نوع مثل url.Values لأن هذا يعني ضرورة اعتماد هذه المكتبة على عملائها.

سنصبح عالقين سريعًا لو لم نجد طريقة لفحص تمثيل قيم الأنواع غير المعروفة. من ثمّ، فإن ما نحتاجه هو الانعكاس.

12.2 reflect.Type و reflect.Value

تقدم حزمة الانعكاس reflect الانعكاس، وهي تعرّف نوعين مهمين هما Type و Value. يمثل Type نوع في Go، وهو واجهة ذات طُرق متعددة للتمييز بين الأنواع وبحث مكوناتها، مثل حقول struct أو مؤشرات وظيفة ما. إن التطبيق المنفرد لـ reflect.Type هو واصف النوع (انظر 7.5)، وهو نفس الكيان الذي يحدد النوع الديناميكي لقيمة الواجهة.

تقبل وظيفة reflect.TypeOf أي {}interface، وتعيد نوعها الديناميكي كـ reflect.Type :

```
t := reflect.TypeOf(3)  // a reflect.Type
fmt.Println(t.String()) // "int" f
mt.Println(t)           // "int"
```

إن استدعاء TypeOf(3) المذكور أعلاه يخصص قيمة 3 لمؤشر {}interface. لقد ذكرنا في القسم 7.5 أن التخصيص من قيمة ملموسة إلى نوع واجهة ينتج تحويل ضمني للواجهة، وهو ما يؤدي لخلق قيمة واجهة من مكونين هما: النوع الديناميكي (dynamic type) وهو نوع المعامل (int)، والقيمة الديناميكية (dynamic value) وهي قيمة المعامل (3).

يعيد reflect.TypeOf يعيد نوعا محددا دائمًا نظرًا لأنه يعيد النوع الديناميكي لقيمة الواجهة. من ثمّ، كمثال، تطبع الشفرة أدناه "os.File*" وليس "io.Writer". ولاحقا، سنرى أن reflect.Type قادر على تمثيل أنواع الواجهة أيضًا.

```
var w io.Writer = os.Stdout
fmt.Println(reflect.TypeOf(w)) // "*os.File"
```

لاحظ أن reflect.Type يرضي واجهة fmt.Stringer. ونظرًا لكون طباعة النوع الديناميكي للواجهة مفيد لاكتشاف الأخطاء وعلاجها وفي الولوج، تقدم fmt.Printf اختصارًا، T%، يستخدم reflect.TypeOf داخليًا:

```
fmt.Printf("%T\n", 3) // "int"
```

إن النوع المهم الآخر في حزمة reflect هو Value. إن reflect.Value يمكن أن يحمل قيمة من أي نوع، وتقبل وظيفة reflect.ValueOf أي {}interface، وتعيد reflect.Value تحتوي على القيمة الديناميكية للواجهة. كما هو الحال مع reflect.TypeOf، تكون نتائج reflect.ValueOf محددة دائمًا، ولكن reflect.Value يمكن أن تحمل قيم واجهة أيضًا:

```
v := reflect.ValueOf(3)    // a reflect.Value
fmt.Println(v)             // "3"
fmt.Printf("%v\n", v)      // "3"
fmt.Println(v.String())    // NOTE: "<int Value>"
```

إن reflect.Value – مثلها مثل reflect.Type – ترضي واجهة fmt.Stringer أيضًا، ولكن ما لم تحمل Value سلسلة، فإن نتيجة طريقة String ستكشف فقط عن النوع. بدلاً من ذلك، استخدم v verb% في حزمة fmt، والذي يعامل reflect.Values معاملة خاصة.

إن استدعاء طريقة Type في Value يعيد نوعها كـ reflect.Value:

```
t := v.Type()              // a reflect.Type
fmt.Println(t.String())    // "int"
```

إن العملية العكسية لـ reflect.ValueOf هي طريقة reflect.Value.Interface. وهي تعيد {}interface تحمل نفس القيمة الملموسة التي تقدمها reflect.Value:

```
v := reflect.ValueOf(3)    // a reflect.Value
x := v.Interface()         // an interface{}
i := x.(int)               // an int
fmt.Printf("%d\n", i)      // "3"
```

إن reflect.Value و {}interface يمكن أن يحملا قيما عشوائية، والفارق بينهما هو أن الواجهة الفارغة تخفي التمثيل والعمليات الداخلية للقيمة التي تحملها، ولا تكشف عن أي من طرقها، وبالتالي ما لم نكن نعلم نوعها الديناميكي، ونستخدم تأكيد النوع للنظر بداخلها (كما فعلنا أعلاه)، لا يوجد سوى القليل مما يمكننا فعله للقيمة بداخلها. على النقيض، تحتوي Value على قيم متعدد لفحص محتوياتها، بغض النظر عن نوعها. لنستخدمها في محاولتنا الثانية مع وظيفة التهيئة العامة، والتي ينطبق عليها format.Any.

سنستخدم طريقة Kind الخاصة بـ reflect.Value لتمييز الحالات بدلاً من استخدام تبديل النوع. وبالرغم من وجود أنواع عديدة لا نهائية، إلا أن هناك عدد محدود من الأصناف (kinds) الخاصة بالنوع: الأنواع الأساسية: Bool و String وكل الأرقام، والأنواع المجمّعة: Array و Struct، والأنواع المرجعية: Chan و Func و Ptr و Slice و Map، وأنواع الواجهة، وأخيرًا Invalid، والتي تعني عدم وجود قيمة على الإطلاق. (إن القيمة الصفرية في reflect.Value ذات صنف Invalid).

```
gopl.io/ch12/format
package format
import (
    "reflect"
    "strconv"
)
// Any formats any value as a string.
func Any(value interface{}) string {
    return formatAtom(reflect.ValueOf(value))
}
// formatAtom formats a value without inspecting its internal structure.
func formatAtom(v reflect.Value) string {
    switch v.Kind() {
    case reflect.Invalid:
        return "invalid"
    case reflect.Int, reflect.Int8, reflect.Int16,
        reflect.Int32, reflect.Int64:
        return strconv.FormatInt(v.Int(), 10)
    case reflect.Uint, reflect.Uint8, reflect.Uint16,
        reflect.Uint32, reflect.Uint64, reflect.Uintptr:
        return strconv.FormatUint(v.Uint(), 10)
    // ...floating-point and complex cases omitted for brevity...
    case reflect.Bool:
        return strconv.FormatBool(v.Bool())
    case reflect.String:
        return strconv.Quote(v.String())
    case reflect.Chan, reflect.Func, reflect.Ptr, reflect.Slice, reflect.Map:
        return v.Type().String() + " 0x" +
            strconv.FormatUint(uint64(v.Pointer()), 16)
    default: // reflect.Array, reflect.Struct, reflect.Interface
        return v.Type().String() + " value"
    }
}
```

تعامل وظيفتنا حتى الآن كل قيمة كشيء غير مرئي بدون هيكل داخلي – وبالتالي تظهر formatAtom. وبالنسبة للأنواع المجمعة (structs و arrays)، والواجهات، فإنها تطبع فقط "نوع" القيمة، أما في الأنواع المرجعية (القنوات، والوظائف، والمؤشرات، والشرائح، والخرائط)، فإنها تطبع النوع، والعنوان المرجعي بالنظام السداسي العشري. إن هذا الوضع ليس مثاليًا، ولكنه يظل بمثابة تحسن كبير، وحيث أن Kind معنية فقط بالتمثيل الضمني، تعمل format.Any مع الأنواع المسماة أيضًا. كمثال:

```
var x int64 = 1
```

```
var d time.Duration = 1 * time.Nanosecond
fmt.Println(format.Any(x))                  // "1"
fmt.Println(format.Any(d))                  // "1"
fmt.Println(format.Any([]int64{x}))         // "[]int64 0x8202b87b0"
fmt.Println(format.Any([]time.Duration{d}))// "[]time.Duration 0x8202b87e0"
```

12.3 Display، طابعة القيمة التكرارية

سنلقي نظرة في الجزء التالي على كيفية تحسين عرض الأنواع المُركبة، بدلاً من محاولة نسخ fmt.Sprint بالضبط، وسنبني وظيفة لكشف الأخطاء تُدعى Display، والتي إذا أُعطيت قيمة معقدة x عشوائيًا، ستطبع بِنية كاملة لتلك القيمة، وتصنف كل عنصر بالمسار الذي وُجِدَت من خلاله. لنبدأ بمثال:

```
e, _ := eval.Parse("sqrt(A / pi)")
Display("e", e)
```

في الاستدعاء أعلاه، كان المُعطى لـ Display هو شجرة بنيوية من مُقيّم التعبير في القسم 7.9. إن ناتج Display موضح أدناه:

```
Display e (eval.call):
e.fn = "sqrt"
e.args[0].type = eval.binary
e.args[0].value.op = 47
e.args[0].value.x.type = eval.Var
e.args[0].value.x.value = "A"
e.args[0].value.y.type = eval.Var
e.args[0].value.y.value = "pi"
```

يجب أن تتجنب كشف الانعكاس في API الخاص بالحزمة كلما أمكن. سنحدد وظيفة display غير مُصدّرة لتقوم بعمل التكرار الحقيقي، ووظيفة Display مُصدّرة، والتي ستكون بمثابة غلاف بسيط حول العمل لتقبل مؤشر {}interface:

```
gopl.io/ch12/display

func Display(name string, x interface{}) {
    fmt.Printf("Display %s (%T):\n", name, x)
    display(name, reflect.ValueOf(x))
}
```

سنستخدم في Display وظيفة formatAtom التي عرفناها سابقًا لطباعة القيمة الابتدائية – الأنواع الأساسية والوظائف والقنوات – ولكننا سنستخدم طُرق reflect.Value لعرض كل مكون في الأنواع المعقدة أكثر بطريقة تكرارية. ومع تراجع التكرار، ستُعَزّز سلسلة path – التي تصف قيمة البدء مبدئيًا (كمثال "e") بحيث تشير إلى كيف وصلنا إلى القيمة الحالية (كمثال: "e.args[0].value").

وحيث أننا لم نعد نتظاهر بتطبيق fmt.Sprint، سنستخدم حزمة fmt لإبقاء مثالنا قصيرًا:

```
func display(path string, v reflect.Value) {
    switch v.Kind() {
    case reflect.Invalid:
        fmt.Printf("%s = invalid\n", path)
    case reflect.Slice, reflect.Array:
        for i := 0; i < v.Len(); i++ {
            display(fmt.Sprintf("%s[%d]", path, i), v.Index(i))
        }
    case reflect.Struct:
        for i := 0; i < v.NumField(); i++ {
            fieldPath := fmt.Sprintf("%s.%s", path, v.Type().Field(i).Name)
            display(fieldPath, v.Field(i))
        }
    case reflect.Map:
        for _, key := range v.MapKeys() {
            display(fmt.Sprintf("%s[%s]", path,
                formatAtom(key)), v.MapIndex(key))
        }
    case reflect.Ptr:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            display(fmt.Sprintf("(*%s)", path), v.Elem())
        }
    case reflect.Interface:
        if v.IsNil() {
            fmt.Printf("%s = nil\n", path)
        } else {
            fmt.Printf("%s.type = %s\n", path, v.Elem().Type())
            display(path+".value", v.Elem())
        }
    default: // basic types, channels, funcs
        fmt.Printf("%s = %s\n", path, formatAtom(v))
    }
}
```

لنناقش الحالات بالترتيب:

الشرائح والمصفوفات (Slices and arrays): إن المنطق الذي يسري على الاثنين واحد. فطريقة Len تعيد عدد عناصر قيمة الشريحة أو المصفوفة، بينما تستعيد Index(i) العنصر في index i، والمنطق مثله مثل reflect.Value أيضًا، حيث يهلع لو كان i خارج الحدود. إن هذه كلها مناظرة لعمليات len(a) و a[i] المدمجة في التسلسلات. تستحث وظيفة display نفسها تكراريًا في كل عنصر في التسلسل، وتضيف إلى الرمز السفلي "[i]" إلى المسار.

بالرغم من أن reflect.Value يحتوي على العديد من الطرق، إلا أن القليل فقط منها يمكن استدعاءه بشكل آمن في أي قيمة معينة. كمثال، يمكن استدعاء طريقة Index في القيمة من صنف Slice أو Array أو String، ولكنها تهلع في حالة أي صنف آخر.

Structs: تقدم طريقة NumField تقريرا بعدد الحقول في struct، وتعيد Field(i) قيمة حقل i-th كـ reflect.Value. تتضمن قائمة الحقول حقولاً تم ترقيتها من حقول مجهولة. لإضافة رمز اختيار الحقل ".f" إلى مسار، يجب أن نحصل على reflect.Type الخاص بـ struct، وندخل اسم حقل i-th الخاص به.

الخرائط (Maps): تُعيد طريقة MapKeys شريحة من reflects.Values، شريحة لكل مفتاح خريطة، ولكن ترتيبها غير مُحدد كما هو الحال عند تكرار خريطة. تعيد MapIndex(key) القيمة المناظرة لـ Key. نحن نضيف الرمز السفلي "[key]" للمسار (نحن نختصر هنا، فنوع مفتاح الخريطة غير محدودة بالأنواع التي تتعامل معها formatAtom بأفضل شكل، وهي المصفوفات والبنيات، كما أن الواجهات يمكن أن تصبح مفاتيح خريطة صحيحة كذلك. إن التمرين 21.1 يوسّع هذه الحالة بحيث تتضمن طباعة المفتاح بالكامل).

المؤشرات (Pointers): تعيد طريقة Elem المتغيرات التي يشير لها مؤشر، وهذا مرة أخرى يشبه reflect.Value. إن هذه العملية ستكون آمنة لو كانت قيمة المؤشر nil، وفي تلك الحالة سيكون صنف النتيجة هو Invalid، ولكننا سنستخدم IsNil لكشف مؤشرات nil صراحة حتى يمكننا طباعة رسالة مناسبة أكثر. سنلحق المسار بـ "*"، ونضعها بين أقواس لتجنب الغموض.

الواجِهات (Interfaces): مرة أخرى، سنستخدم IsNil لاختبار ما إذا كانت الواجهة nil، ولو لم تكن كذلك، فإننا نستعيد قيمتها الديناميكية باستخدام v.Elem()، ونطبع نوعها وقيمتها.

بعد أن اكتملت وظيفة Display لدينا الآن، لنطبقها. إن نوع Movie أدناه هو تنويع طفيف على الموجود في القسم 4.5:

```go
type Movie struct {
    Title, Subtitle string
    Year            int
    Color           bool
    Actor           map[string]string
    Oscars          []string
    Sequel          *string
}
```

لنعلن عن قيمة هذا النوع ونرى ما ستفعله Display بها:

```go
strangelove := Movie{
    Title:    "Dr. Strangelove",
    Subtitle: "How I Learned to Stop Worrying and Love the Bomb",
    Year:     1964,
    Color:    false,
    Actor: map[string]string{
        "Dr. Strangelove":            "Peter Sellers",
        "Grp. Capt. Lionel Mandrake": "Peter Sellers",
        "Pres. Merkin Muffley":       "Peter Sellers",
        "Gen. Buck Turgidson":        "George C. Scott",
```

```
        "Brig. Gen. Jack D. Ripper":  "Sterling Hayden",
        `Maj. T.J. "King" Kong`:      "Slim Pickens",
    },
    Oscars: []string{
        "Best Actor (Nomin.)",
        "Best Adapted Screenplay (Nomin.)",
        "Best Director (Nomin.)",
        "Best Picture (Nomin.)",
    },
}
```

إن الاستدعاء ("strangelove", strangelove) يطبع:

```
Display strangelove (display.Movie):
strangelove.Title = "Dr. Strangelove"
strangelove.Subtitle = "How I Learned to Stop Worrying and Love the Bomb"
strangelove.Year = 1964
strangelove.Color = false
strangelove.Actor["Gen. Buck Turgidson"] = "George C. Scott"
strangelove.Actor["Brig. Gen. Jack D. Ripper"] = "Sterling Hayden"
strangelove.Actor["Maj. T.J. \"King\" Kong"] = "Slim Pickens"
strangelove.Actor["Dr. Strangelove"] = "Peter Sellers"
strangelove.Actor["Grp. Capt. Lionel Mandrake"] = "Peter Sellers"
strangelove.Actor["Pres. Merkin Muffley"] = "Peter Sellers"
strangelove.Oscars[0] = "Best Actor (Nomin.)"
strangelove.Oscars[1] = "Best Adapted Screenplay (Nomin.)"
strangelove.Oscars[2] = "Best Director (Nomin.)"
strangelove.Oscars[3] = "Best Picture (Nomin.)"
strangelove.Sequel = nil
```

يمكننا استخدام Display لعرض الأجزاء الداخلية في أنواع المكتبة، مثل os.File*:

```
Display("os.Stderr", os.Stderr)
// Output:
// Display os.Stderr (*os.File):
// (*(*os.Stderr).file).fd = 2
// (*(*os.Stderr).file).name = "/dev/stderr"
// (*(*os.Stderr).file).nepipe = 0
```

لاحظ أن حتى الحقول غير المُصدّرة ظاهرة للانعكاس. وانتبه إلى أن الناتج المحدد لهذا المثال يمكن أن يتفاوت بين المنصات ويمكن أن يتغير مع مرور الوقت مع تطور المكتبات. (إن هذه الحقول خاصة لهذا السبب). يمكننا حتى تطبيق Display على reflect.Value، ومشاهدتها وهي تتجاوز التمثيل الداخلي لكاشف النوع لـ os.File*. إن ناتج استدعاء Display("rV", reflect.ValueOf(os.Stderr)) ظاهر أدناه، بالرغم من أن عدد الأميال الخاص بك قد يختلف عن هذا:

```
Display rV (reflect.Value):
```

```
(*rV.typ).size = 8
(*rV.typ).hash = 871609668
(*rV.typ).align = 8
(*rV.typ).fieldAlign = 8
(*rV.typ).kind = 22
(*(*rV.typ).string) = "*os.File"
(*(*(*rV.typ).uncommonType).methods[0].name) = "Chdir"
(*(*(*(*rV.typ).uncommonType).methods[0].mtyp).string) = "func() error"
(*(*(*(*rV.typ).uncommonType).methods[0].typ).string) = "func(*os.File) error"
```

لاحظ الفارق بين هذين المثالين:

```
var i interface{} = 3
Display("i", i)
// Output:
// Display i (int):
// i = 3
Display("&i", &i)
// Output:
// Display &i (^interface {}):
// (*&i).type = int
// (*&i).value = 3
```

في المثال الأول، استدعت Display الـ reflect.ValueOf(i)، والتي أعادت قيمة من الصنف Int. وكما ذكرنا في القسم 12.2، تعيد reflect.ValueOf دائمًا قيمة من نوع ملموس حيث أنها تستخلص محتويات قيمة الواجهة.

في المثال الثاني، قامت Display باستدعاء reflect.ValueOf(&i)، والتي أعادت مؤشر إلى i من الصنف Ptr. إن حالة التبديل لـ Ptr تستدعي Elem في هذه القيمة، والتي تعيد Value تمثل المتغير i نفسه، من الصنف Interface. إن Value التي تم الحصول عليها بشكل غير مباشر، مثل هذه، يمكن أن تمثل أي قيمة أيًا كانت، وهذا يتضمن الواجهات. إن وظيفة Display تستدعي نفسها تكراريًا، وهذه المرة .. تطبع مكونات منفصلة للنوع والقيمة الديناميكيين للواجهة.

إن Display، كما تُطبق حاليًا، لن تنتهي أبدًا لو واجهت دورة في رسم بياني لكائن، كما لو أنها دائرة تأكل ذيلها.

```
// a struct that points to itself
type Cycle struct{ Value int; Tail *Cycle }
var c Cycle c = Cycle{42, &c}
Display("c", c)
```

تطبع Display هذا التمدد المتنامي:

```
Display c (display.Cycle):
c.Value = 42
(*c.Tail).Value = 42
```

```
(*(*c.Tail).Tail).Value = 42
(*(*(*c.Tail).Tail).Tail).Value = 42
... ad infinitum...
```

تحتوي العديد من برامج Go على بعض البيانات الدورية على الأقل، وجَعل Display قوية في مواجهة هذه الدورات هو أمر صعب، ويحتاج لحفظ سجلات إضافي لتسجيل مجموعة المراجع التي اتُبِعَت حتى الآن، وهو أمر مكلف كذلك. يتطلب الحل العام خصائص لغوية غير آمنة، كما سنرى في القسم 13.3.

لا تمثل الدورات مشكلة بنفس الخطورة على fmt.Sprint لأنها نادرًا ما تحاول طباعة البنية الكاملة. كمثال، عندما تواجه مؤشر، تكسر التكرار من خلال طباعة القيمة العددية للمؤشر. وقد تصبح عالقة لأنها تحاول طباعة شريحة أو خريطة تحتوي على نفسها كعنصر، ولكن هذه حالات نادرة ولا تستحق الجهد الإضافي المبذول في معالجة الدوران.

تمرين 12.1: وسّع Display بحيث يمكن أن تعرض الخرائط التي مفاتيحها بنيات أو مصفوفات.

تمرين 12.2: اجعل Display آمنة للاستخدام في بنيات البيانات الدورية من خلال تقييد عدد الخطوات التي تقوم بها قبل التخلي عن التكرار. (سنرى في القسم 13.3 طريقة أخرى لكشف الدورات).

12.4 مثال: ترميز تعبيرات S

إن Display هو روتين لكشف الخلل والأخطاء يُستخدم في عرض البيانات المبنية، ولكنه ليس بعيدًا عن ترميز أو صَف (marshal) كائنات Go العشوائية كرسائل في تدوين محمول مناسب للتواصل بين العمليات.

تدعم مكتبة Go القياسية – كما رأينا في القسم 4.5 – مجموعة متنوعة من الصيغ، وهذا يشمل JSON و XML و ASN.1. وهناك تدوين آخر لازال يُستخدم على نطاق واسع هو تعبيرات S (S-expressions)، والتي تُعَد بِنية Lisp. على العكس من التدوينات الأخرى، تعبيرات S غير مدعومة بواسطة مكتبة Go القياسية، وخاصة لأنها ليس لها أي تعريف مقبول عالميًا بالرغم من وجود العديد من المحاولات لتوحيدها، ووجود العديد من التطبيقات.

ستعرّف في هذا القسم حزمة ترمز كائنات Go العشوائية باستخدام تدوين تعبير S الذي يدعم البِنيات التالية:

```
42         integer
"hello"    string (with Go-style quotation)
foo        symbol (an unquoted name)
(1 2 3)    list (zero or more items enclosed in parentheses)
```

إن القيم المنطقية (Booleans) تُرمز عادة باستخدام الرمز t للإشارة إلى (صحيح)، والقائمة الفارغة ()، أو رمز nil للإشارة إلى (خاطئ)، ولكن للتبسيط، سيتجاهل تطبيقنا كل هذا. وهو يتجاهل كذلك القنوات والوظائف، حيث أن

حالتهم معتمة أمام الانعكاس. وهي تتجاهل أرقام الفاصلة العائمة الحقيقية والمعقدة، وتتجاهل الواجهات كذلك. إن إضافة دعم لهم هو موضوع التمرين 12.3.

سنرمز أنواع Go باستخدام تعبيرات S كالتالي. تُرمز الأرقام الصحيحة والسلاسل بطريقة واضحة. أما قيم Nil فتُرمز كرمز nil. وتُرمز المصفوفات والشرائح باستخدام تدوين القائمة.

تُرمز البنيات كقائمة من روابط الحقل، وكل رابط حقل هو قائمة من عنصرين، العنصر الأول (رمز) هو اسم الحقل، والعنصر الثاني هو قيمة الحقل. تُرمز الخرائط كذلك كقائمة من أزواج، وكل زوج هو مفتاح وقيمة مُدخل خريطة واحد. تمثل تعبيرات S عادة قوائم بثنائيات المفتاح/القيمة باستخدام خلايا cons منفردة (key . value) لكل زوج، بدلاً من قائمة من عنصرين، ولكن لتبسيط فك الترميز، سنتجاهل تدوين القائمة المتقطعة.

يتم التركيز بواسطة وظيفة تكرارية واحدة هي encode، والتي تظهر أدناه. إن بنيتها هي نفس بنية Display الموجودة في القسم السابق:

```go
gopl.io/ch12/sexpr

func encode(buf *bytes.Buffer, v reflect.Value) error {
	switch v.Kind() {
	case reflect.Invalid:
		buf.WriteString("nil")
	case reflect.Int, reflect.Int8, reflect.Int16,
		reflect.Int32, reflect.Int64:
		fmt.Fprintf(buf, "%d", v.Int())
	case reflect.Uint, reflect.Uint8, reflect.Uint16,
		reflect.Uint32, reflect.Uint64, reflect.Uintptr:
		fmt.Fprintf(buf, "%d", v.Uint())
	case reflect.String:
		fmt.Fprintf(buf, "%q", v.String())
	case reflect.Ptr:
		return encode(buf, v.Elem())
	case reflect.Array, reflect.Slice: // (value ...)
		buf.WriteByte('(')
		for i := 0; i < v.Len(); i++ {
			if i > 0 {
				buf.WriteByte(' ')
			}
			if err := encode(buf, v.Index(i)); err != nil {
				return err
			}
		}
		buf.WriteByte(')')
	case reflect.Struct: // ((name value) ...)
		buf.WriteByte('(')
		for i := 0; i < v.NumField(); i++ {
			if i > 0 {
				buf.WriteByte(' ')
			}
			fmt.Fprintf(buf, "(%s ", v.Type().Field(i).Name)
			if err := encode(buf, v.Field(i)); err != nil {
				return err
```

```
            }
            buf.WriteByte(')')
        }
        buf.WriteByte(')')
    case reflect.Map: // ((key value) ...)
        buf.WriteByte('(')
        for i, key := range v.MapKeys() {
            if i > 0 {
                buf.WriteByte(' ')
            }
            buf.WriteByte('(')
            if err := encode(buf, key); err != nil {
                return err
            }
            buf.WriteByte(' ')
            if err := encode(buf, v.MapIndex(key)); err != nil {
                return err
            }
            buf.WriteByte(')')
        }
        buf.WriteByte(')')
    default: // float, complex, bool, chan, func, interface
        return fmt.Errorf("unsupported type: %s", v.Type())
    }
    return nil
}
```

إن وظيفة Marshal تلف أداة الترميز في API مشابه لذلك الموجود في الترميزات الأخرى/...الحزم:

```
// Marshal encodes a Go value in S-expression form.
func Marshal(v interface{}) ([]byte, error) {
    var buf bytes.Buffer
    if err := encode(&buf, reflect.ValueOf(v)); err != nil {
        return nil, err
    }
    return buf.Bytes(), nil
}
```

إليك نتيجة تطبيق Marshal على متغير Strangelove من القسم 12.3:

```
 ((Title "Dr. Strangelove") (Subtitle "How I Learned to Stop Worrying and Lo ve
the Bomb") (Year 1964) (Actor (("Grp. Capt. Lionel Mandrake" "Peter Sell ers")
("Pres. Merkin Muffley" "Peter Sellers") ("Gen. Buck Turgidson" "Geor ge C.
Scott") ("Brig. Gen. Jack D. Ripper" "Sterling Hayden") ("Maj. T.J. \ "King\"
Kong" "Slim Pickens") ("Dr. Strangelove" "Peter Sellers"))) (Oscars ("Best
Actor (Nomin.)" "Best Adapted Screenplay (Nomin.)" "Best Director (N omin.)"
"Best Picture (Nomin.)")) (Sequel nil))
```

يظهر الناتج كله في سطر واحد طويل بحد أدنى من المسافات، مما يجعله صعب القراءة. إليك نفس الناتج مهيأ يدويًا وفقًا لأعراف التعبير S. إن كتابة طباعة جميلة للتعبيرات S سيُترك لك كتمرين (يتحداك)، ويتضمن التحميل من gopl.io نسخة بسيطة:

```
 ((Title "Dr. Strangelove")
(Subtitle "How I Learned to Stop Worrying and Love the Bomb")
(Year 1964)
(Actor (("Grp. Capt. Lionel Mandrake" "Peter Sellers")
("Pres. Merkin Muffley" "Peter Sellers")
("Gen. Buck Turgidson" "George C. Scott")
("Brig. Gen. Jack D. Ripper" "Sterling Hayden")
("Maj. T.J. \"King\" Kong" "Slim Pickens")
("Dr. Strangelove" "Peter Sellers")))
(Oscars ("Best Actor (Nomin.)"
"Best Adapted Screenplay (Nomin.)"
"Best Director (Nomin.)"
"Best Picture (Nomin.)"))
(Sequel nil))
```

ستدخل sexpr.Marshal في حلقة للأبد لو تم استدعاءها ببيانات دورية مثلها مثل وظائف fmt.Print و json.Marshal و Display.

سنوضح في القسم 12.6 تطبيق وظيفة فك ترميز تعبير S المُناظِر، ولكن قبل أن نتطرق لهذا، سنحتاج أولاً لفهم كيف يمكن استخدام الانعكاس لتحديث متغيرات البرنامج.

تمرين 12.3: طبق الحالات المفقودة في وظيفة encode. قم بترميز القيم المنطقية (booleans) كـ t و nil، وأرقام الفاصلة العائمة باستخدام تدوين Go، وأرقام معقدة مثل 1-2i كـ #C(1.0 2.0). يمكن ترميز الواجهات كزوج من نوع الاسم وقيمة، كمثال(("[]int" (1 2 3))، ولكن انتبه إلى أن هذا التدوين مبهم: قد تعيد طريقة reflect.Type.String نفس السلسلة لأنواع مختلفة.

تمرين 12.4: تعديل encode لتطبيق تعبير S بالنمط الموضح أعلاه.

تمرين 12.5: تعديل encode لحذف JSON بدلاً من تعبيرات S. اختبر أداة الترميز الخاصة بك باستخدام أداة فك الشفرات القياسية، json.Unmarshal.

تمرين 12.6: عدّل encode، بحيث لا ترمز حقلاً قيمته هو القيمة الصفرية لنوعه، واعتبار هذا نوع من التحسين.

تمرين 12.7: اصنع بث API لأداة فك ترميز تعبير S، بإتباع نمط json.Decoder (انظر 4.5).

12.6 ضبط المتغيرات باستخدام reflect.Value

لقد "فسّر" الانعكاس حتى الآن القيم الموجودة في برنامجها بطرق متعددة. ولكن الهدف من هذا القسم هو "تغييرها".

تذكّر أن بعض تعبيرات Go مثل x و x.f[1] و p* تشير إلى متغيرات، ولكن هناك تعبيرات أخرى مثل x + 1 و f(2) لا تشير لها. إن المتغير هو موقع تخزين قابل للعنونة (addressable) يحتوي على قيمة، ويمكن تحديث قيمته من خلال هذا العنوان.

هناك اختلاف مشابه ينطبق على reflect.Values. بعضها قابل للعنونة وبعضها ليس كذلك. انظر الإعلانات التالية:

```
x := 2                   // value   type      variable?
a := reflect.ValueOf(2)  // 2       int       no
b := reflect.ValueOf(x)  // 2       int       no
c := reflect.ValueOf(&x) // &x      *int      no
d := c.Elem()            // 2       int       yes (x)
```

إن القيمة داخل a غير قابلة للعنونة، فهي مجرد نسخة من الرقم الصحيح 2، ونفس الشيء ينطبق على b. إن القيمة داخل c هي أيضا غير قابلة للعنونة، فهي نسخة من قيمة المؤشر x&. في الواقع، لا يوجد reflect.Value تعيدها reflect.ValueOf(x) قابلة للعنونة. ولكن d المشتقة من c من خلال إزالة مرجعية المؤشر بداخلها، تشير إلى متغير، وبالتالي تعتبر قابلة للعنونة. يمكننا استخدام هذه الطريقة – أي استدعاء reflect.ValueOf(&x).Elem() للحصول على Value قابلة للعنونة لأي متغير x.

يمكننا أن نسأل reflect.Value ما إذا كانت قابلة للعنونة أم لا من خلال طريقة CanAddr فيها:

```
fmt.Println(a.CanAddr()) // "false"
fmt.Println(b.CanAddr()) // "false"
fmt.Println(c.CanAddr()) // "false"
fmt.Println(d.CanAddr()) // "true"
```

نحن نحصل على reflect.Value قابلة للعنونة كلما وجهناها بشكل غير مباشر عبر مؤشر، حتى لو بدأنا من Value غير قابلة للعنونة. إن كل القواعد المعتاد للعنونة لها نظير في الانعكاس. كمثال، حيث أن تعبير فهرسة الشريحة e[i] يتبع مؤشر ضمنيًا، فإنه يُعَد قابلاً للعنونة حتى لو لم يكن التعبير e قابلاً للعنونة. وبالمثل، يشير reflect.ValueOf(e).Index(i) إلى متغير، وبالتالي يُعتبر قابلاً للعنونة حتى لو لم يكن reflect.ValueOf(e) كذلك.

إن استعادة المتغير من reflect.Value يتطلب ثلاث خطوات. الأولى، هي استدعاء Addr()، والتي تعيد Value تحمل مؤشر للمتغير، والخطوة الثانية هي استدعاء Interface() لهذه الـ Value، والتي تعيد قيمة interface{} تحتوي على مؤشر. الخطوة الثالثة والأخيرة هي أننا لو كنا نعرف نوع المتغير، يمكننا استخدم تأكيد النوع لاستعادة محتويات الواجهة كمؤشر عادي. ويمكننا حينها تحديث المتغير من خلال المؤشر:

```
x := 2
d := reflect.ValueOf(&x).Elem()      //    d refers to the variable x
px := d.Addr().Interface().(*int)    //    px := &x
```

```
*px =3                          //    x = 3
fmt.Println(x)                  //    "3"
```

أو، يمكننا تحديث المتغير المشار إليه عن طريق reflect.Value قابلة للعنونة مباشرة، وبدون استخدام مؤشر، من خلال استدعاء طريقة reflect.Value.Set:

```
d.Set(reflect.ValueOf(4))
fmt.Println(x) // "4"
```

إن نفس الفحوص الخاصة بالقابلية للإسناد (assignability) التي تُجرى بواسطة المترجم تتم في زمن التشغيل بواسطة طُرق Set. كما نلاحظ في التشفير أعلاه، يمتلك كل من المتغير والقيمة النوع int، ولكن لو كان المتغير int64، فإن البرنامج سيهلع، لذا من الضروري التأكد من أن القيمة قابلة للإسناد إلى نوع المتغير:

```
d.Set(reflect.ValueOf(int64(5))) // panic: int64 is not assignable to int
```

وبالطبع فإن استدعاء Set في reflect.Value غير قابلة للعنونة يؤدي للهلع أيضًا:

```
x := 2
b := reflect.ValueOf(x)
b.Set(reflect.ValueOf(3)) // panic: Set using unaddressable value
```

هناك متغيرات لـ Set متخصصة في مجموعات معينة من الأنواع الأساسية: SetInt و SetUint و SetString و SetFloat، إلخ:

```
d := reflect.ValueOf(&x).Elem()
d.SetInt(3)
fmt.Println(x) // "3"
```

إن هذه الطرق أكثر تساهلاً إلى حد ما، فـ SetInt كمثال ستنجح طالما أن نوع المتغير هو رقم صحيح موقّع بشكل ما، أو حتى لو كان النوع المسمى هو رقم صحيح موقّع، ولو كانت القيمة كبيرة جدًا فسيتم اقتطاعها بهدوء حتى تصبح ملائمة. ولكن انتبه، استدعاء SetInt في reflect.Value التي تشير إلى متغير interface{} سيؤدي للهلع، بالرغم من أن Set سينجح:

```
x := 1
rx := reflect.ValueOf(&x).Elem()
rx.SetInt(2)                       // OK, x = 2
rx.Set(reflect.ValueOf(3))         // OK, x = 3
rx.SetString("hello")              // panic: string is not assignable to int
rx. Set(reflect.ValueOf("hello"))  // panic: string is not assignable to int

var y interface{}
ry := reflect.ValueOf(&y).Elem()
```

```
ry. SetInt(2)                 // panic: SetInt called on interface Value
ry.Set(reflect.ValueOf(3))      // OK, y = int(3)
ry.SetString("hello")           // panic: SetString called on interface Value
ry.Set(reflect.ValueOf("hello")) // OK, y = "hello"
```

عندما طبقنا Display على os.Stdout، وجدنا أن الانعكاس يمكن أن يقرأ قيم حقول البنية غير المُصدّرة التي لا يمكن الدخول لها وفقًا لقواعد اللغة التقليدية، مثل حقل fd int في بنية os.File في منصة مثل Unix. مع ذلك، لا يمكن للانعكاس تحديث مثل هذه القيم.

```
stdout := reflect.ValueOf(os.Stdout).Elem() // *os.Stdout, an os.File var
fmt.Println(stdout.Type())// "os.File"
fd := stdout.FieldByName("fd")
fmt.Println(fd.Int())      // "1"
fd.SetInt(2)               // panic: unexported field
```

تسجل reflect.Value القابلة للعنونة ما إذا كان تم الحصول عليها من خلال تجاوز حقل بنية غير مُصدّر أم لا، ولو كان الأمر كذلك، فإنها لا تسمح بالتعديل. من ثمّ، لا يُعتبر CanAddr الفحص المناسب للاستخدام عادة قبل ضبط متغير. إن طريقة CanSet ذات الصلة تقدم تقرير عما إذا كانت reflect.Value قابلة للعنونة وقابلة للضبط أم لا:

```
fmt.Println(fd.CanAddr(), fd.CanSet()) // "true false"
```

12.6 مثال: فك ترميز تعبيرات-S

مقابل كل وظيفة Marshal تقدمها حزم encoding/... الخاصة بالمكتبة القياسية، هناك وظيفة Unmarshal مناظرة لها تقوم بفك الترميز. على سبيل المثال، كما رأينا في القسم 4.5، لو حصلنا على شريحة بايت تحتوي على بيانات مرمزة بـ JSON لنوع Movie الخاص بنا (انظر 12.3)، يمكننا فك ترميزه كالتالي:

```
data := []byte{/* ... */}
var movie Movie
err := json.Unmarshal(data, &movie)
```

تستخدم وظيفة Unmarshal الانعكاس لتعديل حقول متغير movie الموجود، وتخلق خرائط وبنيات وشرائح جديدة يحددها النوع Movie، ومحتوى البيانات الواردة.

لنطبّق الآن وظيفة Unmarshal بسيطة لتعبيرات S، تناظر وظيفة json.Unmarshal القياسية المستخدمة أعلاه، وتعكس وظيفة sexpr.Marshal التي استخدمناها سابقًا. يجب أن ننبهك إلى أن التطبيق القوي والعام يحتاج شفرة أكبر بكثير مما يمكن وضعه في هذا المثال، الطويل بما يكفي في وضعه الحالي، لذا قمنا بالعديد من الاختصارات. لقد قدمنا

فقط مجموعة فرعية محدودة من تعبيرات S ولم نتعامل مع الأخطاء ببراعة كافية. إن الشفرة تهدف إلى توضيح الانعكاس وليس التحليل.

يستخدم lexer نوع Scanner من حزمة text/scanner لتقسيم تيار مُدخلات إلى تسلسل من الرموز مثل التعليقات والمُعرّفات وحروف السلسلة، والحروف العددية. إن طريقة Scan في الماسح الضوئي تطور الماسح ونوع صنع الرمز التالي، وهو النوع rune. تتكون معظم الرموز – مثل '(' – من حرف رونيّ (rune) واحد، ولكن حزمة text/scanner تمثل أصناف الرموز متعددة الحروف مثل Ident و String و Int باستخدام قيمة سلبية صغيرة من النوع rune. بعد استدعاء Scan التي تعيد واحدة من أصناف الرمز هذه، تعيد طريقة TokenText الخاصة بالماسح الضوئي نص الرمز.

نظرًا لكون المحلل (parser) التقليدي قد يحتاج لفحص الرمز الحالي عدّة مرات، ولكن بما أن طريقة Scan تقدم الماسح، سنقوم بلف الماسح في نوع مساعد يُسمى lexer، وهو يتتبع ما يجري في الرمز الذي أعادته Scan.

```
gopl.io/ch12/sexpr

type lexer struct {
    scan  scanner.Scanner
    token rune // the current token
}
func (lex *lexer) next()        { lex.token = lex.scan.Scan() }
func (lex *lexer) text() string { return lex.scan.TokenText() }
func (lex *lexer) consume(want rune) {
    if lex.token != want { // NOTE: Not an example of good error handling.
        panic(fmt.Sprintf("got %q, want %q", lex.text(), want))
    }
    lex.next()
}
```

لننتقل إلى المحلل الآن. يتكون المحلل من وظيفتين أساسيتين. الأولى هي read، وهي تقرأ التعبير S الذي يبدأ بالرمز الحالي ويحدث المتغير المشار له بـ v reflect.Value القابل للعنونة.

```
func read(lex *lexer, v reflect.Value) {
    switch lex.token {
    case scanner.Ident:
        // The only valid identifiers are
        // "nil" and struct field names.
        if lex.text() == "nil" {
            v.Set(reflect.Zero(v.Type()))
            lex.next()
            return
        }
    case scanner.String:
        s, _ := strconv.Unquote(lex.text()) // NOTE: ignoring errors
        v.SetString(s)
        lex.next()
        return
    case scanner.Int:
        i, _ := strconv.Atoi(lex.text()) // NOTE: ignoring errors
```

```
        v.SetInt(int64(i))
        lex.next()
        return
    case '(':
        lex.next()
        readList(lex, v)
        lex.next() // consume ')'
        return
    }
    panic(fmt.Sprintf("unexpected token %q", lex.text()))
}
```

تستخدم تعبيرات S الخاصة بنا المُعرّفات لهدفين مختلفين، وهما أسماء حقل بنية، وقيمة nil الخاصة بالمؤشر. تتعامل وظيفة read مع الحالة الأخيرة فقط. عندما تواجه "nil" scanner.Ident، تحدد قيمة v بالقيمة الصفرية لنوعها باستخدام وظيفة reflect.Zero. أما بالنسبة لأي مُعرّف آخر، فإنها تقدم تقرير بالخطأ. إن وظيفة readList التي سنراها خلال لحظات، تتعامل مع المُعرّفات المستخدمة مثل أسماء حقل بنية.

يشير رمز(token) '(' إلى بداية القائمة. وتقوم الوظيفة الثانية، readList، بفك ترميز قائمة إلى متغير من النوع المركب – مثل خريطة أو بنية أو شريحة أو مصفوفة – سيعتمد هذا على نوع متغير go الذي نشغله حاليًا. وفي كل حالة، ستستمر الحلقة في تحليل العناصر حتى تواجه قوس إغلاق مطابق، ')'، كشفته وظيفة endList.

إن الجزء المثير للاهتمام هو التكرار. إن أبسط حالة هي مصفوفة، حتى يظهر الإغلاق ')'. نحن نستخدم Index للحصول على متغير لكل عنصر مصفوفة، وعمل استدعاء تكراري لـ read لشغلها. وكما هو الحال في العديد من حالات الخطأ الأخرى، لو كانت البيانات المُدخلة تجعل أداة فك الترميز تفهرس ما هو أبعد من نهاية المصفوفة، فإن الأداة تهلع. تُستخدم طريقة مشابهة مع الشرائح، باستثناء أننا يجب أن نصنع متغيرا جديدا لكل عنصر، ونَشغَله، ثم نضيفه للشريحة.

إن حلقات البنيات (structs) والخرائط يجب أن تحلل القائمة الفرعية (لقيمة المفتاح) في كل تكرار. وبالنسبة للبنيات، المفتاح هو رمز يُعرف الحقل. وكما هو الحال في حالة المصفوفات، نحصل على المتغير الموجود لحقل البنية باستخدام FieldByName، وعمل استدعاء تكراري لشغله. بالنسبة للخرائط، يمكن أن يكون المفتاح من أي نوع، وكما هو الحال في حالة الشرائح، نصنع متغير جديد، ونشغله تكراريًا، وفي النهاية ندخل ثنائي key/value الجديد في الخريطة.

```
func readList(lex *lexer, v reflect.Value) {
    switch v.Kind() {
    case reflect.Array: // (item ...)
        for i := 0; !endList(lex); i++ {
            read(lex, v.Index(i))
        }
    case reflect.Slice: // (item ...)
        for !endList(lex) {
            item := reflect.New(v.Type().Elem()).Elem()
            read(lex, item)
            v.Set(reflect.Append(v, item))
        }
```

```
    case reflect.Struct: // ((name value) ...)
        for !endList(lex) {
            lex.consume('(')
            if lex.token != scanner.Ident {
                panic(fmt.Sprintf("got token %q, want field name", lex.text()))
            }
            name := lex.text()
            lex.next()
            read(lex, v.FieldByName(name))
            lex.consume(')')
        }
    case reflect.Map: // ((key value) ...)
        v.Set(reflect.MakeMap(v.Type()))
        for !endList(lex) {
            lex.consume('(')
            key := reflect.New(v.Type().Key()).Elem()
            read(lex, key)
            value := reflect.New(v.Type().Elem()).Elem()
            read(lex, value)
            v.SetMapIndex(key, value)
            lex.consume(')')
        }
    default:
        panic(fmt.Sprintf("cannot decode list into %v", v.Type()))
    }
}
func endList(lex *lexer) bool {
    switch lex.token {
    case scanner.EOF:
        panic("end of file")
    case ')':
        return true
    }
    return false
}
```

أخيرًا، نلف المحلل في الوظيفة المصدرة Unmarshal، الموضحة أدناه، والتي تخفي بعض الأجزاء الخشنة في التطبيق. إن الأخطاء التي واجهناها أثناء التقييم ينتج عنها هلع، وبالتالي تستخدم Unmarshal استدعاء مؤجّل للتعافي من الهلع (انظر 5.10)، وإعادة رسالة خطأ بدلاً من ذلك.

```
// Unmarshal parses S-expression data and populates the variable
// whose address is in the non-nil pointer out.
func Unmarshal(data []byte, out interface{}) (err error) {
    lex := &lexer{scan: scanner.Scanner{Mode: scanner.GoTokens}}
    lex.scan.Init(bytes.NewReader(data))
    lex.next() // get the first token
    defer func() {
        // NOTE: this is not an example of ideal error handling.
        if x := recover(); x != nil {
            err = fmt.Errorf("error at %s: %v", lex.scan.Position, x)
        }
    }()
    read(lex, reflect.ValueOf(out).Elem())
    return nil
}
```

إن تطبيق جودة-الإنتاج (production-quality) لا يجب أن يحدث به هلع أبدًا مع أي مدخَل، ويجب أن يقدم تقرير بالخطأ المعلوماتي في حالة كل خطأ، وربما مع رقم للسطر أو الإزاحة (offset). بالرغم من ذلك، نحن نأمل أن يوصل هذا المثال فكرة ما عما يحدث تحت سطح حزمة مثل encoding/json، وكيف يمكنك استخدام الانعكاس لشَغْل بنيات البيانات.

تمرين 12.8: إن وظيفة sexpr.Unmarshal، مثلها مثل json.Marshal، تحتاج إلى مُدخل كامل في شريحة البايت قبل التمكّن من البدء في فك ترميزها. حدد نوع sexpr.Decoder يسمح – مثل json.Decoder – بفك ترميز تسلسل قيم من io.Reader. غيّر sexpr.Unmarshal لاستخدام هذا النوع الجديد.

تمرين 12.9: اكتب API معتمد على رمز (token) لفك ترميز تعبيرات S، بإتباع نمط xml.Decoder (انظر 7.14). ستحتاج إلى خمس أنواع من الرموز هم: Symbol و String و Int و StartList و EndList.

تمرين 12.10: وسّع sexpr.Unmarshal للتعامل مع القيم المنطقية (Booleans)، وأعداد الفاصلة العائمة، والواجهات المرمزة بواسطة حلك للتمرين 12.3 (تلميح: لفك ترميز الواجهات، ستحتاج إلى رسم خريطة من اسم كل نوع مدعوم إلى reflect.Type الخاص به).

12.7 الوصول لوسوم حقل البنية

استخدمنا في القسم 4.5 وسوم حقل البنية لتعديل ترميز JSOM لقيم بنية Go. يسمح لنا وسم حقل JSON أن نختار أسماء حقل بديل، ونكتب ناتج الحقول الفارغة. سنرى في هذا القسم كيف ندخل لوسوم الحقل باستخدام الانعكاس.

إن أول شيء تقوم به معظم وظائف مداول HTTP في خادم الويب هو استخلاص مؤشرات الطلب إلى متغيرات محلية. سنعرّف وظيفة وسيلة هي params.Unpack، وهي تستخدم وسوم حقل البنية لجعل كتابة مداولات HTTP (انظر 7.7) مريحة أكثر.

أولا، سنوضح كيف تُستخدم. إن وظيفة search أدناه هي مداول HTTP. وهي تُعرّف متغير اسمه data من نوع بنية مجهولة، تتوافق حقوله مع مؤشرات طلب HTTP. تحدد وسوم حقل البنية أسماء المؤشر، والتي عادة ما تكون قصيرة وغامضة لأن المساحة صغيرة في URL. تَشْغَل وظيفة Unpack البنية من الطلب، بحيث يمكن الوصول للمؤشرات بشكل مريح ومع النوع المناسب.

```
gopl.io/ch12/search

import "gopl.io/ch12/params"
```

```
// search implements the /search URL endpoint.
func search(resp http.ResponseWriter, req *http.Request) {
    var data struct {
        Labels     []string `http:"l"`
        MaxResults int      `http:"max"`
        Exact      bool     `http:"x"`
    }
    data.MaxResults = 10 // set default
    if err := params.Unpack(req, &data); err != nil {
        http.Error(resp, err.Error(), http.StatusBadRequest) // 400
        return
    }
    // ...rest of handler...
    fmt.Fprintf(resp, "Search: %+v\n", data)
}
```

تقوم وظيفة Unpack أدناه بثلاث أمور. الأول، أنها تستدعي ()req.ParseForm لتحليل الطلب، وبعدها يحتوي req.Form على كل المؤشرات بعض النظر عما إذا كان عميل HTTP استخدم طريقة طلب GET أم POST.

بعد ذلك، تبني Unpack خريطة من الاسم الفعال (effective) لكل حقل إلى المتغير الخاص بهذا الحقل. قد يختلف الاسم الفعال عن الاسم الفعلي لو كان الحقل يحتوي على وسم. إن طريقة Field في reflect.Type تعيد reflect.StructField يقدم معلومات حول نوع كل حقل، مثل اسمه ونوعه ووسمه المحتمل. إن حقل Tag هو reflect.StructTag، وهو نوع سلسلة يقدم طريقة GET لتحليل واستخلاص السلسلة الفرعية الخاصة بمفتاح محدد، مثل "...":http في هذه الحالة.

```
gopl.io/ch12/params

// Unpack populates the fields of the struct pointed to by ptr
// from the HTTP request parameters in req.
func Unpack(req *http.Request, ptr interface{}) error {
    if err := req.ParseForm(); err != nil {
        return err
    }
    // Build map of fields keyed by effective name.
    fields := make(map[string]reflect.Value)
    v := reflect.ValueOf(ptr).Elem() // the struct variable
    for i := 0; i < v.NumField(); i++ {
        fieldInfo := v.Type().Field(i) // a reflect.StructField
        tag := fieldInfo.Tag           // a reflect.StructTag
        name := tag.Get("http")
        if name == "" {
            name = strings.ToLower(fieldInfo.Name)
        }
        fields[name] = v.Field(i)
    }
    // Update struct field for each parameter in the request.
    for name, values := range req.Form {
        f := fields[name]
        if !f.IsValid() {
            continue // ignore unrecognized HTTP parameters
        }
        for _, value := range values {
            if f.Kind() == reflect.Slice {
```

```
                elem := reflect.New(f.Type().Elem()).Elem()
                if err := populate(elem, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
                f.Set(reflect.Append(f, elem))
            } else {
                if err := populate(f, value); err != nil {
                    return fmt.Errorf("%s: %v", name, err)
                }
            }
        }
    }
    return nil
}
```

أخيرًا، تقوم Unpack بتكرار أزواج name/value في مؤشرات HTTP وتحدث حقول البنية المناظرة. تذكر أن نفس اسم المؤشر يمكن أن يظهر أكثر من مرة. لو حدث هذا، وكان الحقل عبارة عن شريحة، فإن كل قيم هذا المؤشر تتراكم في الشريحة. بخلاف ذلك، يُعاد الكتابة على الحقل باستمرار بحيث يكون للقيمة الأخيرة فيه فقط أي تأثير.

إن وظيفة populate تعتني بضبط حقل v واحد (أو عنصر واحد في حقل شريحة) من قيمة مؤشر. ولكن بالنسبة للآن، ستدعم فقط السلاسل، والأعداد الصحيحة الموقعة، والقيم المنطقية (Booleans). إن دعم أنواع أخرى متروك كتمرين.

```
func populate(v reflect.Value, value string) error {
    switch v.Kind() {
    case reflect.String:
        v.SetString(value)
    case reflect.Int:
        i, err := strconv.ParseInt(value, 10, 64)
        if err != nil {
            return err
        }
        v.SetInt(i)
    case reflect.Bool:
        b, err := strconv.ParseBool(value)
        if err != nil {
            return err
        }
        v.SetBool(b)
    default:
        return fmt.Errorf("unsupported kind %s", v.Type())
    }
    return nil
}
```

لو أضفنا معالج الخادم إلى خادم الويب، فقد تكون هذه جلسة تقليدية:

```
$ go build gopl.io/ch12/search
$ ./search &
$ ./fetch 'http://localhost:12345/search'
Search: {Labels:[] MaxResults:10 Exact:false}
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming'
Search: {Labels:[golang programming] MaxResults:10 Exact:false}
```

```
$ ./fetch 'http://localhost:12345/search?l=golang&l=programming&max=100'
Search: {Labels:[golang programming] MaxResults:100 Exact:false}
$ ./fetch 'http://localhost:12345/search?x=true&l=golang&l=programming' Search:
{Labels:[golang programming] MaxResults:10 Exact:true}
$ ./fetch 'http://localhost:12345/search?q=hello&x=123' x: strconv.ParseBool:
parsing "123": invalid syntax
$ ./fetch 'http://localhost:12345/search?q=hello&max=lots' max:
strconv.ParseInt: parsing "lots": invalid syntax
```

تمرين 12.11: اكتب وظيفة Pack مناظرة. لو مُنحت قيمة بنية، فإن Pack يجب أن تعيد URL يدمج قيم المؤشر من البنية.

تمرين 12.12: وسّع تدوين وسم الحقل للتعبير عن متطلبات صلاحية المؤشر. كمثال، قد تحتاج السلسلة لأن تكون عنوان بريد إلكتروني صحيح أو رقم بطاقة ائتمان، وقد يحتاج الرقم الصحيح لأن يكون رمز بريدي صحيح في الولايات المتحدة. عدّل Unpack لفحص هذه المتطلبات.

تمرين 12.13: عدّل أداة ترميز التعبير S (انظر 12.4) وأداة فك ترميزه (انظر 12.6)، بحيث يقدّرون وسم sexprfield بطريقة مشابهة لـ encoding/json (انظر 4.5).

12.8 عرض الطُرق الخاصة بنوع

إن مثالنا الأخير على الانعكاس يستخدم reflect.Type لطباعة نوع قيمة عشوائية وتعداد طُرقها:

```go
gopl.io/ch12/methods

// Print prints the method set of the value x.
func Print(x interface{}) {
	v := reflect.ValueOf(x)
	t := v.Type()
	fmt.Printf("type %s\n", t)
	for i := 0; i < v.NumMethod(); i++ {
		methType := v.Method(i).Type()
		fmt.Printf("func (%s) %s%s\n", t, t.Method(i).Name,
			strings.TrimPrefix(methType.String(), "func"))
	}
}
```

إن كل من reflect.Type و reflect.Value يحتويان على طريقة اسمها Method، وكل استدعاء لـ t.Method(i) يعيد مثال على reflect.Method، ونوع بنية يصف اسم ونوع طريقة منفردة. إن كل استدعاء لـ v.Method(i) يعيد reflect.Value تمثل قيمة طريقة (انظر 6.4)، أي أن الطريقة مقيدة بمستلِمَها. إن استخدام طريقة reflect.Value.Call (والتي لا نمتلك

مساحة كافية لتوضيحها هنا)، تجعل من الممكن استدعاء Values الصنف Func كما هو الحال في هذا المال، ولكن هذا البرنامج يحتاج فقط إلى الـ Type الخاص به.

إليك الطرق التي تنتمي للنوعين: time.Duration و*strings.Replacer :

```
methods.Print(time.Hour)
// Output:
// type time.Duration // func (time.Duration) Hours() float64
// func (time.Duration) Minutes() float64
// func (time.Duration) Nanoseconds() int64
// func (time.Duration) Seconds() float64
// func (time.Duration) String() string

methods.Print(new(strings.Replacer))
// Output:
// type *strings.Replacer
// func (*strings.Replacer) Replace(string) string
// func (*strings.Replacer) WriteString(io.Writer, string) (int, error)
```

12.9 تنبيه أخير

إن هناك الكثير من الأمور المتعلقة بانعكاس API أكثر مما يمكننا تقديمه هنا، ولكن الأمثلة السابقة تمنحك فكرة عما هو ممكن. إن الانعكاس هو أداة قوية ومعبرة، ولكن يجب استخدامه بحرص، وهذا لثلاثة أسباب.

السبب الأول هو أن الشفرة المعتمدة على الانعكاس يمكن أن تكون هشة، فكل خطأ يجعل المترجم يقدم خطأ خاص بالنوع، ستجد هناك طريقة مناظرة لإساءة استخدام الانعكاس، ولكن رغم أن المترجم يقدم تقرير بالخطأ في زمن البناء، إلا أن خطأ الانعكاس يُقدّم خلال التنفيذ كحالة هلع، وغالبًا ما يحدث هذا بعد فترة طويلة من كتابة البرنامج أو حتى بعد فترة من بدء تشغيله.

لو كانت وظيفة readList (انظر 12.6) ستقرأ سلسلة من المُدخل بينما تَشْغَل متغير من النوع int، فإن استدعاء reflect.Value.SetString سيهلع. إن معظم البرامج التي تستخدم الانعكاس معرضة لأخطاء شبيهة، ويجب التصرف بحرص شديد لمتابعة النوع، والقابلية للعنونة، والقابلة للضبط، وكل reflect.Value.

إن أفضل طريقة لتجنب هذه الهشاشة هي ضمان أن استخدام الانعكاس مغلف تمامًا داخل حزمتك، ولو أمكن، تجنب reflect.Value لصالح أنواع محددة في API حزمتك، وتقييد المُدخلات بالقيم القانونية. لو لم يكن هذا ممكنًا، قم بإجراء فحوص ديناميكية إضافية قبل كل عملية بها مخاطرة. أحد الأمثلة من المكتبة القياسية، عندما تطبق fmt.Printf فِعل

على معامل غير مناسب، فإنها لا تهلع بشكل غامض، بل تطبع رسالة خطأ معلوماتية. سيظل هناك خلل بالبرنامج، ولكن سيكون من السهل تشخيصه حينها.

```
fmt.Printf("%d %s\n", "hello", 42) // "%!d(string=hello) %!s(int=42)"
```

يقلل الانعكاس أيضًا سلامة ودقة إعادة التصنيع الآلية وأدوات التحليل لأنها لا تستطيع تحديد معلومات النوع أو الاعتماد عليها.

السبب الثاني لتجنب الانعكاس هو أن الأنواع تعمل كشكل من أشكال التوثيق، وعمليات الانعكاس لا تخضع لفحص النوع الثابت، مما يجعل الشفرة العاكسة القوية صعبة الفهم. وثّق الأنواع المتوقعة بحرص شديد دائما وكذلك ثوابت الوظائف الأخرى التي تقبل {}interface و reflect.Value.

إن السبب الثالث هو أن الوظائف المعتمدة على الانعكاس قد تكون أبطأ بدرجة أو اثنين من الشفرة المخصصة لنوع محدد. لا ترتبط أغلبية الوظائف في البرامج التقليدية بالأداء الكلي، لذا لا بأس من استخدام الانعكاس عندما يجعل البرنامج أوضح. إن الاختبار مناسب بشكل خاص للانعكاس لأن معظم الاختبارات تستخدم مجموعات بيانات صغيرة، ولكن من الأفضل تجنب الانعكاس في الوظائف الموجودة على المسار الحَرِج.

13- البرمجة منخفضة المستوى

يضمن تصميم Go عددًا من خصائص الأمان التي تحُد من حدوث أخطاء في برنامج go. وخلال التجميع والترجمة، يكشف التحقق من النوع معظم محاولات تطبيق عملية ما على قيمة غير مناسبة لهذا النوع، كمثال، طرح سلسلة من سلسلة أخرى. إن القواعد الصارمة لتحويلات النوع تمنع الدخول المباشر للأجزاء الداخلية من الأنواع المدمجة مثل السلاسل والخرائط والشرائح والقنوات.

تضمن الفحوص الديناميكية إغلاق البرنامج فورًا مع تقديم رسالة خطأ معلوماتي كلما حدثت عملية ممنوعة من عمليات الأخطاء التي لا يمكن كشفها إحصائيًا - مثل الوصول إلى خارج نطاق المصفوفة أو حالات استرجاع مؤشر nil. إضافة إلى أن إدارة الذاكرة الآلية (جمع القمامة) تتخلص من أخطاء "الاستخدام بعد تحرير الذاكرة" (use after free)، وكذلك من معظم تسريبات الذاكرة.

لا يمكن لبرامج go الوصول للعديد من تفاصيل التطبيق، ولا يوجد أي طريقة لاكتشاف مخطط ذاكرة نوع مُجمع مثل struct أو شفرة آلة خاصة بوظيفة معينة، أو هوية خيط نظام التشغيل الذي يعمل عليه جو-روتين الحالي. ينقل مُجدْوِل Go الـ روتينات جو بحُرية من خيط إلى آخر. ويحدد المؤشر المتغير دون الكشف عن العناوين الرقمية للمتغير. ويمكن أن تتغير العناوين مع تحريك جامع القمامة للمتغيرات، كما تُحدّث المؤشرات بشفافية.

إن هذه الخصائص معًا تجعل برامج Go - خاصة التي تفشل - قابلة للتوقع أكثر، وأقل غموضًا من برامج لغة C، اللغة الجوهرية منخفضة المستوى.

إن إخفاء التفاصيل الضمنية يجعل برامج Go محمولة (portable) إلى حد كبير، حيث أن دلالات اللغة مستقلة بشكل كبير عن أي مترجم محدد أو نظام تشغيل أو بنية CPU. (وإن كانت ليست مستقلة تمامًا: بعض التفاصيل تتسرب مثل حجم كلمة المعالج، وترتيب تقييم تعبيرات معينة، ومجموعة قيود التطبيق التي يفرضها المترجم).

قد نختار من حين لآخر خسارة بعض هذه الضمانات المفيدة من أجل تحقيق أعلى أداء ممكن، وتشغيل البيني للمكتبات المكتوبة بلغات أخرى، أو تطبيق وظيفة لا يمكن التعبير عنها بلغة go خالصة.

سنرى في هذا الفصل كيف تجعلنا حزمة unsafe نخطو خارج قواعدنا المعتادة، وكيفية استخدام أداة cgo لخلق روابط Go مع مكتبات C واستدعاءات نظام التشغيل.

لا يجب استخدام المناهج الموصوفة في هذا الفصل باندفاع، فبدون الانتباه الدقيق للتفاصيل، يمكن أن تؤدي لأعطال غير متوقعة وغير ملحوظة وغير محلية يعرفها مُبرمجو لغة C جيدًا لسوء الحظ. إن استخدام حزمة unsafe يُبطل ضمان جو بالتوافق مع الإصدارات المستقبلية، لأن من السهل الاعتماد على تفاصيل التطبيق غير المحددة التي يمكن أن تتغير بشكل غير متوقع، سواء أكانت متعمدة أم غير متعمدة.

إن حزمة unsafe سحرية، فبالرغم من أنها تبدو حزمة عادية، وتُستورد بالطريقة المعتادة، إلا أن المترجم هو الذي يطبقها فعليا. وهي تقدم مدخلاً لعدد من خصائص اللغة المدمجة غير المتاحة في العادة لأنها تكشف تفاصيل مخطط ذاكرة Go.إن تقديم هذه التفاصيل كحزمة منفصلة يجعل المناسبات النادرة التي تكون هناك حاجة لهم فيها بارزة أكثر. كذلك، قد تقيد بعض البيئات استخدام حزمة unsafe لأسباب متعلقة بالأمن.

تُستخدم حزمة unsafe بشكل موسع ضمن الحزم منخفضة المستوى مثل runtime و os و syscall و net التي تتفاعل مع نظام التشغيل، ولكن لا تحتاجها البرامج العادية إلا نادرًا.

13.1 unsafe.Sizeof و Alignof و Offsetof

تقدم وظيفة unsafe.Sizeof تقارير بحجم تمثيل عاملها بالبايت، وهو ما يمكن أن يكون تعبير من أي نوع، والتعبير لا يُقيّم. كذلك فإن استدعاء Sizeof هو تعبير ثابت عن النوع uintptr، لذا يمكن استخدام النتيجة كبُعد لنوع مصفوفة، أو لحساب الثوابت الأخرى.

```
import "unsafe"
fmt.Println(unsafe.Sizeof(float64(0))) // "8"
```

تقدم Sizeof تقريبًا بحجم الجزء الثابت في كل بنية بيانات فقط، مثل المؤشر وطول السلسلة، ولكنها لا تقدم حجم الأجزاء غير المباشرة مثل محتويات السلسلة. إن الأجزاء التقليدية في كل أنواع Go غير المجموعة موضحة أدناه، بالرغم من أن الأحجام الدقيقة قد تتفاوت تبعًا لسلسلة الأدوات. ولجعل البرامج مجموعة أكثر، قدمنا أحجام الأنواع المرجعية (أو الأنواع التي تحتوي على مراجع) من حيث الكلمات، حيث الكلمة هي 4 بايت في منصة 32 بِت و 8 بايت في منصة 64 بِت.

تحمّل أجهزة الكمبيوتر القيم من الذاكرة وتخزنها بأكبر قدر من الكفاءة عندما تكون هذه القيم بمحاذاة بعضها. على سبيل المثال، عنوان قيمة ذات نوع من اثنين بايت مثل int16 يجب أن يكون رقم زوجي، وعنوان قيمة من 4 بايت مثل rune يجب أن يكون من مضاعفات الرقم أربعة، وعنوان قيمة من 8 بايت، مثل float64 و unit64 أو مؤشر 64 بِت، يجب أن

يكون من مضاعفات الرقم ثمانية. إن متطلبات المحاذاة الخاصة بالمضاعفات الأعلى غير معتادة حتى في أنواع البيانات الأكبر مثل complex128.

لهذا السبب، سيكون حجم قيمة نوع مجمع (بنية أو مصفوفة) هو على الأقل مجموع أحجام حقوله أو عناصره، ولكنه قد يكون أكبر بسبب وجود "ثقوب" (holes). إن الثقوب هي مساحات غير مستخدمة يضيفها المترجم لضمان أن الحقل أو العنصر الحالي يحاذي بداية البنية أو المصفوفة بشكل مناسب.

```
Type                          Size
bool                          1 byte
intN, uintN, floatN, complexN N/8 bytes (for example, float64 is 8 bytes)
int, uint, uintptr            1 word
*T                                   1 word
string                        2 words (data, len)
[]T                           3 words (data, len, cap)
map                           1 word
func                          1 word
chan                          1 word
interface                     2 words (type, value)
```

لا تضمن مواصفات اللغة أن ترتيب إعلان الحقول هو الترتيب الذي سيوضعون به في الذاكرة، لذا نظريًا يمتلك المترجم حرية إعادة ترتيبهم، بالرغم من أنه لا يوجد مترجم يفعل هذا حتى لحظة كتابتنا لهذه السطور. ولو كانت أنواع حقول struct ذات أحجام مختلفة، فقد يكون من الأفضل إعلان الحقول بالترتيب الذي يضعهم بأقرب شكل لبعضهم توفيرًا للمساحة. إن البنيات (structs) الثلاثة أدناه بها بعض الحقول، ولكن البنية الأولى تتطلب ذاكرة أكثر ب 50% من البنيتين الأخرتين:

```
                                        // 64-bit 32-bit
struct{ bool; float64; int16 }  // 3 words 4 words
struct{ float64; int16; bool }  // 2 words 3 words
struct{ bool; int16; float64 }  // 2 words 3 words
```

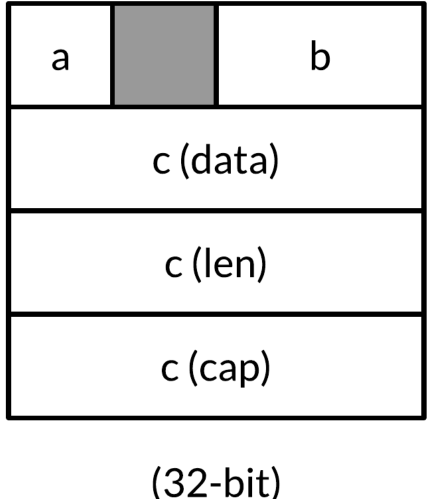
إن تفاصيل خوارزمية المحاذاة تتجاوز نطاق هذا الكتاب، وهي لا تستحق بالتأكيد القلق بشأن كل بنية، ولكن التعبئة الكفء يمكن أن تجعل بنيات البيانات التي تخصص بشكل متكرر مضغوطة أكثر وبالتالي ستصبح أسرع.

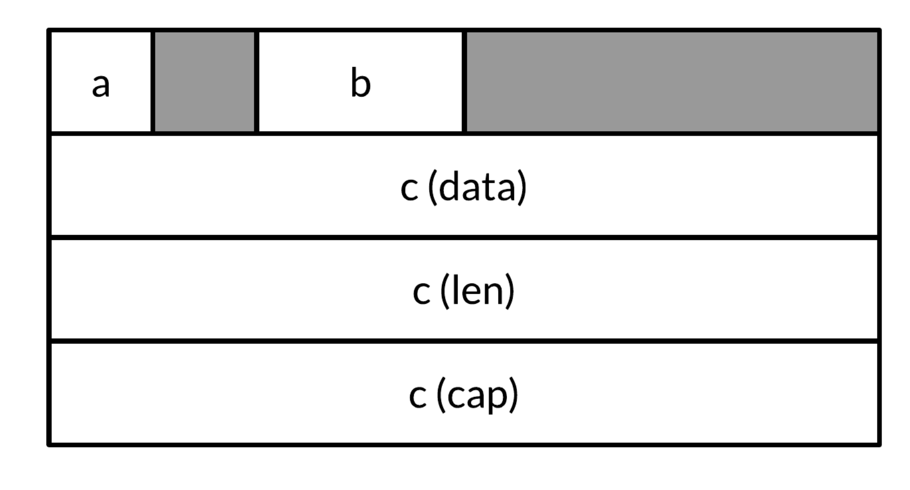
تقدم وظيفة unsafe.Alignof المحاذاة المطلوبة من نوع المعطى الخاص بها. ومثلها مثل Sizeof، يمكن تطبيقها على تعبير من أي نوع، لأنها تقدم ثابت (constant). عادة ما تكون القيمة المنطقية (Boolean) والأنواع الرقمية بمحاذاة حجمهم (بحد أقصى 8 بايت)، وكل الأنواع الأخرى تكون بمحاذاة الكلمات.

إن وظيفة unsafe.Offsetof، يجب أن يكون عاملها هو منتقي الحقل x.f، وهي تحسب إزاحة الحقل f بالنسبة لبداية بنية التغليف x الخاصة به، مراعية الثقوب إن وُجِدت.

يوضح لشكل 13.1 متغير البنية x ومخطط ذاكرته في تطبيقات 32 بِت و 64 بِت تقليدية. إن المناطق الرمادية هي ثقوب:

```
var
x struct {
    a bool
    b int16
    c []int
}
```





الشكل 13.1: ثقوب في بنية.

يوضح الجدول أدناه نتائج تطبيق وظائف unsafe الثلاثة على x نفسه وعلى كل حقل من حقوله الثلاثة:

منصة 32 بِت تقليدية:

```
Sizeof(x)   = 16     Alignof(x)   = 4
Sizeof(x.a) = 1      Alignof(x.a) = 1 Offsetof(x.a) = 0
Sizeof(x.b) = 2      Alignof(x.b) = 2 Offsetof(x.b) = 2
Sizeof(x.c) = 12     Alignof(x.c) = 4 Offsetof(x.c) = 4
```

منصة 64 بِت تقليدية:

```
Sizeof(x)   = 32     Alignof(x)   = 8
Sizeof(x.a) = 1      Alignof(x.a) = 1  Offsetof(x.a) = 0
Sizeof(x.b) = 2      Alignof(x.b) = 2  Offsetof(x.b) = 2
Sizeof(x.c) = 24     Alignof(x.c) = 8  Offsetof(x.c) = 8
```

إن هذه الوظائف غير آمنة في الحقيقة بالرغم من أسماءها، وقد تكون مفيدة في فهم مخطط الذاكرة الخام في برنامج عند تحسين المساحة.

13.2 unsafe.Pointer

إن معظم أنواع المؤشرات تُكتب T*، وتعني "مؤشر إلى متغير من النوع T". إن النوع unsafe.Pointer هو نوع مميز من المؤشرات يمكن أن يحمل عنوان أي متغير. لا يمكننا بالطبع التجول بشكل غير مباشر عبر unsafe.Pointer باستخدام p* لأننا لا نعرف ما هو النوع الخاص بهذا التعبير. إن unsafe.Pointers مثلها مثل المؤشرات العادية، قابلة للمقارنة ويمكن مقارنتها مع nil، وهي القيمة الصفرية للنوع.

يمكن تحويل مؤشر T* العادي إلى unsafe.Pointers، ويمكن تحويل unsafe.Pointers مرة أخرى إلى مؤشر عادي، وليس بالضرورة لنفس النوع T*. كمثال، عند تحويلنا مؤشر float64* إلى uint64*، سنفحص نمط البِت الخاص بمتغير الفاصلة العائلة:

```
package math
func Float64bits(f float64) uint64 { return *(*uint64)(unsafe.Pointer(&f)) }
fmt.Printf("%#016x\n", Float64bits(1.0)) // "0x3ff0000000000000"
```

من خلال معرفة المؤشر، يمكننا أيضا تحديث نمط البت. وهذا غير مضر لمتغير من نوع النقطة العائمة لأن كل أنماط البت مسموح بها، ولكن بشكل عام، تحويلات unsafe.Pointer تسمح لنا بكتابة قيم عشوائية إلى الذاكرة وبالتالي إفساد نظام الأنواع.

إن كل مؤشر من نوع unsafe.Pointer يمكن تحويله إلى uintptr يمكنه حفظ قيمة المؤشر العددية، مما يسمح لنا بإجراء عمليات حسابية في العناوين. (تذكر من الفصل الثالث أن uintptr هو عدد بلا إشارة كبيرة بما يكفي لتمثيل أي عنوان.) إن هذا التحويل يمكن أيضا أن يطبق بشكل عكسي، ولكن مجددا، التحويل من uintptr إلى unsafe.Pointer يمكن أن يفسد نظام الأنواع لأن ليس كل الأرقام هي عناوين صالحة.

وبالتالي فإن العديد من قيم unsafe.Pointer هي وسطاء لتحويل المؤشرات العادية إلى عناوين رقمية خامة والعكس. إن المثال أدناه يأخذ عنوان من متغير x ، ويضيف إزاحة حقل b، ويحول نتيجة العنوان إلى int16* ومن خلال ذلك يحدث قيمة x.b:

```
gopl.io/ch13/unsafeptr

var x struct {
```

```
    a bool
    b int16
    c []int
}

// equivalent to pb := &x.b
pb := (*int16)(unsafe.Pointer(
    uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)))
*pb = 42

fmt.Println(x.b) // "42"
```

بالرغم من أن الصياغة مرهِقة – ربما لا يكون هذا أمرًا سيئًا حيث أن هذه الخصائص يجب استخدامها عند الضرورة فقط – لا تحاول تقديم متغيرات مؤقتة من النوع uintptr لتقسيم الخطوط. إن الشفرة غير صحيحة:

```
// NOTE: subtly incorrect!
tmp := uintptr(unsafe.Pointer(&x)) + unsafe.Offsetof(x.b)
pb := (*int16)(unsafe.Pointer(tmp))
#pb = 42
```

إن السبب دقيق جدًا، وهو أن بعض جامعي القمامة ينقلون المتغيرات في أنحاء الذاكرة لتقليل التجزئة أو لحفظ السجلات. يُعرف جامعو القمامة من هذا النوع بـ "moving GCs". عندما يتحرك المتغير، يجب تحديث كل المؤشرات التي تحمل عنوان موقعه القديم بحيث تشير إلى الموقع الجديد. ومن منظور جامع القمامة، تعتبر unsafe.Pointer مؤشرًا، وبالتالي يجب أن تتغير قيمتها مع حركة المتغير، ولكن uintptr هو مجرد رقم، وبالتالي لا يجب أن تتغير قيمته. إن الشفرة غير الصحيحة أعلاه تخفي مؤشر عن جامع القمامة في المتغير tmp الذي لا يُعد مؤشرًا. وبحلول وقت تنفيذ العبارة الثانية، سيكون المتغير x قد تحرك غالبًا، والرقم في tmp لن يظل عنوان &x.b . تضرب العبارة الثالثة موقع الذاكرة العشوائية بقيمة 42.

إن هناك مجموعة كبيرة من التنويعات الباثولوجية على هذه السِمة (theme). بعد تنفيذ هذه العبارة:

```
pT := uintptr(unsafe.Pointer(new(T))) // NOTE: wrong!
```

لن يعود هناك مؤشرات تشير للمتغير الذي صنعته new، والتالي سيكون من حق جامع القمامة إعادة تدوير مخزونه عند إكمال هذه العبارة، وسيحتوي pT بعدها على العنوان الذي كان فيه المتغير سابقًا ولكنه لم يعُد موجودًا فيه الآن.

لا يوجد تطبيق حالي لـ Go يستخدم جامع قمامة متحرك (بالرغم من أن التطبيقات المستقبلية يمكن أن تفعل هذا)، ولكن ليس هذا سببًا للرضا عن الوضع الراهن: نسخ Go الحالية تحرك بعض المتغيرات داخل الذاكرة. تذكر ما قلناه في القسم 5.2 عن أن رصّات goroutine تنمو حسب الحاجة. عندما يحدث هذا، يمكن أن يتغير موقع كل المتغيرات في

الرصة القديمة وتنتقل إلى رصة جديدة أكبر، وبالتالي لا يمكننا الاعتماد على بقاء القيمة الرقمية لعنوان المتغير دون تغيير خلال عمره كله.

لا يوجد سوى القليل من التوجيهات - في وقت كتابة هذا الكتاب - حول ما يمكن لمبرمجي Go الاعتماد عليه بعد تحويل unsafe.Pointer إلى uintptr (انظر إصدار Go رقم 7192)، لذا نوصي بشدة بأن تفترض عدم وجود أي شيء سوى الحد الأدنى. عامِل كل قيم uintptr كما لو كان تحتوي على عنوان "سابق" للمتغير، وقلل عدد العمليات بين تحويل unsafe.Pointer إلى uintptr واستخدام هذا الـ uintptr لأدنى حد. في أول مثال قدمناه أعلاه، ظهر في سطر واحد العمليات الثلاثة كلها - أي التحويل إلى uintptr، وإضافة إزاحة الحقل، وإعادة التحويل.

عند استدعاء وظيفة مكتبة تعيد uintptr، كتلك المُقدَّمة أدناه من حزمة reflect، يجب تحويل النتيجة فورًا إلى unsafe.Pointer لضمان استمرارها في الإشارة إلى نفس المتغير.

```
package reflect
func (Value) Pointer() uintptr
func (Value) UnsafeAddr() uintptr
func (Value) InterfaceData() [2]uintptr // (index 1)
```

13.3 مثال: التكافؤ العميق

توضح وظيفة DeepEqual من حزمة reflect ما إذا كانت قيمتان متساويتان "بعمق" أم لا. تقارن DeepEqual القيم الأساسية كما لو كانت تُقارَن بواسطة عامل == المدمج، أما بالنسبة للقيم المركبة، فإنها تعترضهم بشكل متكرر، وتقارن العناصر المناظرة لهم. تُستخدم هذه الوظيفة على نطاق واسع في الاختبارات لأنها تعمل على أي زوج من القيم، حتى القيم غير القابلة للمقارنة مع ==. تستخدم الاختبارات التالية DeepEqual لمقارنة قيمتيّ []string:

```
func TestSplit(t *testing.T) {
    got := strings.Split("a:b:c", ":")
    want := []string{"a", "b", "c"};
    if !reflect.DeepEqual(got, want) { /* ... */ }
}
```

بالرغم من أن DeepEqual مريحة، إلا أن الفروق الخاصة بها يمكن أن تبدو عشوائية. كمثال، لا تعتبر خريطة nil مساوية لخريطة non-nil الفارغة، ولا تعتبر شريحة nil مساوية لشريحة non-nil الفارغة:

```
var a, b []string = nil, []string{}
fmt.Println(reflect.DeepEqual(a, b)) // "false"

var c, d map[string]int = nil, make(map[string]int)
```

```
fmt.Println(reflect.DeepEqual(c, d)) // "false"
```

سنعرّف في هذا القسم وظيفة Equal التي تقارن القيم العشوائية، وهي مثلها مثل DeepEqual، فهي تقارن الشرائح والخرائط بناء على عناصرها، ولكن على العكس من DeepEqual، تعتبر شريحة (أو خريطة) nil مساوية لـ non-nil الفارغة أيضًا. إن التكرار الأساسي للمعطيات يمكن أن يتم باستخدام الانعكاس، باستخدام طريقة مشابهة لبرنامج Display الذي رأيناه في القسم 12.3. كما هو معتاد، سنعرّف الوظيفة غير المُصدرة، equal، من أجل التكرار. لا تقلق بشأن المعامل seen الآن. في كل زوج من القيم x و y يتم مقارنتهم، تتحقق equal من أن كلاهما صحيح (أو غير صحيح) وتتحقق مما إذا كانا يمتلكان نفس النوع. تُعرّف نتيجة الوظيفة كمجموعة من حالات التبديل التي تقارن قيمتين من نفس النوع. ونتيجة لأسباب متعلقة بالمساحة، حذفنا العديد من الحالات نظرًا لكون النمط أصبح مألوفًا الآن.

```
gopl.io/ch13/equal
func equal(x, y reflect.Value, seen map[comparison]bool) bool {
    if !x.IsValid() || !y.IsValid() {
        return x.IsValid() == y.IsValid()
    }
    if x.Type() != y.Type() {
        return false
    }
    // ...cycle check omitted (shown later)...
    //!-
    //!+cyclecheck
    // cycle check
    if x.CanAddr() && y.CanAddr() {
        xptr := unsafe.Pointer(x.UnsafeAddr())
        yptr := unsafe.Pointer(y.UnsafeAddr())
        if xptr == yptr {
            return true // identical references
        }
        c := comparison{xptr, yptr, x.Type()}
        if seen[c] {
            return true // already seen
        }
        seen[c] = true
    }
    //!-cyclecheck
    //!+
    switch x.Kind() {
    case reflect.Bool:
        return x.Bool() == y.Bool()
    case reflect.String:
        return x.String() == y.String()
    // ...numeric cases omitted for brevity...
    //!-
    case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32,
        reflect.Int64:
        return x.Int() == y.Int()
    case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32,
        reflect.Uint64, reflect.Uintptr:
        return x.Uint() == y.Uint()
    case reflect.Float32, reflect.Float64:
        return x.Float() == y.Float()
    case reflect.Complex64, reflect.Complex128:
```

```
        return x.Complex() == y.Complex()
    //!+
    case reflect.Chan, reflect.UnsafePointer, reflect.Func:
        return x.Pointer() == y.Pointer()
    case reflect.Ptr, reflect.Interface:
        return equal(x.Elem(), y.Elem(), seen)
    case reflect.Array, reflect.Slice:
        if x.Len() != y.Len() {
            return false
        }
        for i := 0; i < x.Len(); i++ {
            if !equal(x.Index(i), y.Index(i), seen) {
                return false
            }
        }
        return true
    // ...struct and map cases omitted for brevity...
    //!-
    case reflect.Struct:
        for i, n := 0, x.NumField(); i < n; i++ {
            if !equal(x.Field(i), y.Field(i), seen) {
                return false
            }
        }
        return true
    case reflect.Map:
        if x.Len() != y.Len() {
            return false
        }
        for _, k := range x.MapKeys() {
            if !equal(x.MapIndex(k), y.MapIndex(k), seen) {
                return false
            }
        }
        return true
        //!+
    }
    panic("unreachable")
}
```

كالعادة، نحن لا نكشف استخدام الانعكاس في API، وبالتالي فإن الوظيفة المُصدرة Equal يجب أن تستدعي reflect.ValueOf لأجل معطياتها:

```
// Equal reports whether x and y are deeply equal.
func Equal(x, y interface{}) bool {
    seen := make(map[comparison]bool)
    return equal(reflect.ValueOf(x), reflect.ValueOf(y), seen)
}

type comparison struct {
    x, y unsafe.Pointer
    t reflect.Type
}
```

لضمان إنهاء الخوارزمية حتى لبنيات البيانات الدورية، يجب أن تسجل أزواج المتغيرات التي قارنتها بالفعل لتجنب مقارنتها لمرة ثانية. تخصص Equal مجموعة من بنيات المقارنة comparison، يحمل كل منها عنوان اثنين من المتغيرات (التي تُمثّل كقيم unsafe.Pointer)، ونوع المقارنة. نحن نحتاج لتسجيل النوع إضافة إلى العناوين لأن المتغيرات المختلفة يمكن أن يكون لها نفس العنوان. على سبيل المثال، لو كان x و y كلاهما مصفوفات، و x و x[0] لهما نفس العنوان، وكذلك y و y[0] نفس العنوان، فمن المهم التمييز بين ما إذا كنا نقارن بين x و y أم بين x[0] و y[0].

بمجرد أن تؤكد equal أن معطياتها من نفس النوع، وقبل أن تنفذ التبديل، ستتحقق مما إذا كانت تقارن متغيرين شوهدا بالفعل، ولو كانا كذلك فستحذف التكرار.

```
// cycle check
if x.CanAddr() && y.CanAddr() {
    xptr := unsafe.Pointer(x.UnsafeAddr())
    yptr := unsafe.Pointer(y.UnsafeAddr())
    if xptr == yptr {
        return true // identical references
    }
    c := comparison{xptr, yptr, x.Type()}
    if seen[c] {
        return true // already seen
    }
    seen[c] = true
}
```

هذه هي وظيفة Equal أثناء عملها:

```
fmt.Println(Equal([]int{1, 2, 3}, []int{1, 2, 3}))        // "true"
fmt.Println(Equal([]string{"foo"}, []string{"bar"}))      // "false"
fmt.Println(Equal([]string(nil), []string{}))             // "true"
fmt.Println(Equal(map[string]int(nil), map[string]int{})) // "true"
```

بل إنها تعمل أيضًا على المُدخلات الدورية المشابهة لتلك الناتجة عن وظيفة Display التي قدمناها في القسم 12.3 لتصبح عالقة في حلقة:

```
// Circular linked lists a -> b -> a and c -> c.
type link struct {
    value string
    tail *link
}
a, b, c := &link{value: "a"}, &link{value: "b"},&link{value: "c"}
a.tail, b.tail, c.tail = b, a, c
fmt.Println(Equal(a, a))   // "true"
fmt.Println(Equal(b, b))  // "true"
fmt.Println(Equal(c, c))  // "true"
fmt.Println(Equal(a, b))  // "false"
```

```
fmt.Println(Equal(a, c)) // "false"
```

تمرين 13.1: عرّف وظيفة مقارنة عميقة تعتبر الأرقام (من أي نوع) متساوية لو كانت تختلف عن بعضها بأقل من جزء من المليون.

تمرين 13.2: اكتب وظيفة تذكر ما إذا كانت معطياتها هي بنية بيانات دورية أم لا.

13.4 استدعاء شفرة C باستخدام cgo

قد يحتاج برنامج Go إلى استخدام تعريف عتاد (hardware driver) مُطبّق بلغة C، أو الاستعلام من قاعدة بيانات مدمجة مطبقة بلغة C++، أو استخدام بعض روتينات الجبر الخطية المُطبقة بـ Fortran. كانت C هي لغة البرمجة السائدة لفترة طويلة، والعديد من الحزمة السائدة التي تُستخدم على نطاق واسع تصدّر API متوافقة على C، بغض النظر عن لغة تطبيق الحزم.

سنبني في هذا القسم برنامج ضغط بيانات بسيط يستخدم cgo، وهي أداة لإنشاء ارتباطات Go من أجل وظائف C. يُطلق على مثل هذه الأدوات "واجهات الوظيفة الخارجية" (FFIs) (أو foreign-function interfaces)، وليست cgo الأداة الوحيدة المُستخدمة في برامج Go، فـ SWIG (swig.org) هي أداة أخرى تقدم خصائص أكثر تعقيدًا للتكامل مع فئات C++، ولكننا لن نقدمها هنا.

إن الشجرة الفرعية ...compress/ من المكتبة القياسية تقدم ضواغط (compressors) ومخففات انضغاط (decompressors) لخوارزميات الضغط الشائعة، بما في ذلك LZW (التي يستخدمها أمر الضغط Unix)، و DEFLATE (التي يستخدمها أمر GNU gzip). تتفاوت تفاصيل APIs الخاصة بهذه الحزم بشكل طفيف جدًا، ولكنها تقدم غلاف لـ io.Reader يخفف الضغط عن البيانات التي تُقرأ منه. كمثال:

```
package gzip // compress/gzip
func NewWriter(w io.Writer) io.WriteCloser
func NewReader(r io.Reader) (io.ReadCloser, error)
```

إن خوارزمية bzip2، التي تعتمد على تحويل Burrows-Wheeler الراقي، تعمل بسرعة أبطأ من gzip، ولكن ينتج عنها ضغط أفضل بنسبة كبيرة. وتقدم حزمة compress/bzip2 مخفف انضغاط لـ bzip2، ولكن لم تقدم الحزمة أي ضاغط حتى هذه اللحظة. إن بناء واحد من الصفر هو مهمة كبيرة، ولكن هناك تطبيق C مفتوح المصدر موثق جيدًا وعالي الأداء، وهو حزمة libbzip2 من bzip.org.

لو كانت مكتبة C صغيرة، فسننقلها إلى Go خالصة وحسب، ولو كان أداءها غير ضروري بالنسبة لأهدافنا، فسيكون من الأفضل أن نستدعي برنامج C كعملية فرعية مساعدة باستخدام حزمة os/exec. سيكون من المنطقي أكثر استخدام cgo وتغليفها (wrap) عندما تحتاج لاستخدام مكتبة معقدة ذات أداء حرج وذات API ضيقة بلغة C. سنقدم في بقية هذا الفصل مثالاً على هذا.

سنحتاج من حزمة libbzip2 C إلى نوع بنية bz_stream، والتي تحمل صوانات مُدخَل ومُخرج، وثلاث وظائف C، وهي: BZ2_bzCompressInit الذي يخصص صوانات البث، و BZ2_bzCompress، الذي يضغط البيانات من صوان المُدخل إلى صوان المُخرج، و BZ2_bzCompressEnd، الذي يحرر الصوانات. (لا تقلق بشأن آليات حزمة libbzip2، فهدف هذا المثال هو توضيح كيف تتوافق الأجزاء معًا).

سنستدعي وظائف BZ2_bzCompressInit و BZ2_bzCompressEnd C من Go مباشرة، ولكن بالنسبة لـ BZ2_bzCompress، سنعرّف وظيفة الغلاف في C، لتوضيح كيف يتم الأمر. إن ملف مصدر C أدناه يتواجد جنبًا إلى جنب مع شفرة Go في حزمتنا.

```
gopl.io/ch13/bzip

/* This file is gopl.io/ch13/bzip/bzip2.c,         */
/* a simple wrapper for libbzip2 suitable for cgo. */
#include <bzlib.h>
int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen) {
  s->next_in = in;
  s->avail_in = *inlen;
  s->next_out = out;
  s->avail_out = *outlen;
  int r = BZ2_bzCompress(s, action);
  *inlen -= s->avail_in;
  *outlen -= s->avail_out;
  s->next_in = s->next_out = NULL;
  return r;
}
```

لننتقل الآن إلى شفرة Go وسنجد الجزء الأول منها أدناه. إن إعلان "import "C مميز، ولا يوجد حزمة C، ولكن هذا الاستيراد يجعل go build يعالج الملف مسبقًا باستخدام أداة cgo قبل أن يراه مترجم Go.

```
// Package bzip provides a writer that uses bzip2 compression (bzip.org).
package bzip
/*
#cgo CFLAGS: -I/usr/include
#cgo LDFLAGS: -L/usr/lib -lbz2
#include <bzlib.h>
#include <stdlib.h>
bz_stream* bz2alloc() { return calloc(1, sizeof(bz_stream)); }
```

```
int bz2compress(bz_stream *s, int action,
                char *in, unsigned *inlen, char *out, unsigned *outlen);
void bz2free(bz_stream* s) { free(s); }
*/
import "C"
import (
    "io"
    "unsafe"
)
type writer struct {
    w      io.Writer // underlying output stream
    stream *C.bz_stream
    outbuf [64 * 1024]byte
}
// NewWriter returns a writer for bzip2-compressed streams.
func NewWriter(out io.Writer) io.WriteCloser {
    const blockSize = 9
    const verbosity = 0
    const workFactor = 30
    w := &writer{w: out, stream: C.bz2alloc()}
    C.BZ2_bzCompressInit(w.stream, blockSize, verbosity, workFactor)
    return w
}
```

تنتج cgo خلال مرحلة المعالجة المسبقة حزمة مؤقتة تحتوي على إعلانات go المناظرة لكل وظائف C والأنواع التي استخدمها الملف، مثل C.bz_stream و C.BZ2_bzCompressInit. تكتشف أداة cgo هذه الأنواع من خلال استدعاء مترجم C بطريقة خاصة لتفسير محتويات التعليق الذي يسبق إعلان الاستيراد.

قد يحتوي التعليق على أوامر توجيهية #cgo تحدد خيارات إضافة لسلسلة أدوات C. تساهم قيم CFLAGS و LDFLAGS بمعطيات إضافية لأوامر المترجم والموصّل، بحيث يمكنها تحديد ملف ترويسة bzlib.h، ومكتبة أرشيف bzlib.h. يفترض المثال أن هذه تُثبت تحت usr/ في نظامك. قد تحتاج لتعديل أو حذف تلك الأعلام لأجل التثبيت.

تستدعي NewWriter وظيفة C BZ2_bzCompressInit لبدء صوانات البث. ويتضمن النوع Writer صوان آخر سيُستخدم لتفريغ صوان مُخرَج مخفف الضغط.

إن طريقة Write الموضحة أدناه تغذي البيانات غير المضغوطة في الضاغط، وتستدعي الوظيفة bz2compress في حلقة حتى يتم استهلاك جميع البيانات. لاحظ أن برنامج Go يمكن أن يدخل لأنواع C مثل bz_stream و Char و uint، وكذلك برامج C مثل bz2compress، بل يمكن أن تدخل حتى إلى ماكرو معالِج C المسبق الشبيه بالكائن مثل BZ_RUN، وكل هذا من خلال تدوين C.x . إن نوع C.uint مختلف عن نوع uint الخاص بلغة Go، حتى لو كان كلاهما يمتلك نفس العرض.

```
func (w *writer) Write(data []byte) (int, error) {
    if w.stream == nil {
        panic("closed")
```

```
    }
    var total int // uncompressed bytes written
    for len(data) > 0 {
        inlen, outlen := C.uint(len(data)), C.uint(cap(w.outbuf))
        C.bz2compress(w.stream, C.BZ_RUN,
            (*C.char)(unsafe.Pointer(&data[0])), &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        total += int(inlen)
        data = data[inlen:]
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return total, err
        }
    }
    return total, nil
}
```

يمرر كل تكرار للحلقة عنوان وطول الجزء الباقي من الباقي إلى bz2compress، وكذلك عنوان وسعة w.outbuf. يُمرّر متغيرّ الطول بواسطة عناوينهما وليس قيمهما، حتى تتمكن وظيفة C من تحديثهما لتوضيح مقدار البيانات غير المضغوطة المستهلكة، ومقدار البيانات المضغوطة الناتجة. إن كل قطعة من البيانات المضغوطة تُكتب في io.Writer الضمنية بعد ذلك.

إن بنية طريقة Close مشابهة لـ Write، وتستخدم حلقة للتخلص من أي بيانات مضغوطة باقي من تيار صوان مُخرج البث.

```
// Close flushes the compressed data and closes the stream.
// It does not close the underlying io.Writer.
func (w *writer) Close() error {
    if w.stream == nil {
        panic("closed")
    }
    defer func() {
        C.BZ2_bzCompressEnd(w.stream)
        C.bz2free(w.stream)
        w.stream = nil
    }()
    for {
        inlen, outlen := C.uint(0), C.uint(cap(w.outbuf))
        r := C.bz2compress(w.stream, C.BZ_FINISH, nil, &inlen,
            (*C.char)(unsafe.Pointer(&w.outbuf)), &outlen)
        if _, err := w.w.Write(w.outbuf[:outlen]); err != nil {
            return err
        }
        if r == C.BZ_STREAM_END {
            return nil
        }
    }
}
```

بعد الانتهاء، تقوم Close باستدعاء BZ2_bzCompressEnd. لتحرير صوانات البث باستخدام defer لضمان حدوث هذا في كل مسارات العودة (return). لكن عند هذه النقطة لن يكون من الآمن عمل إسناد مؤشري (dereference) لمؤشر

w.stream. لذا نقوم بضبطه على nil – كرد فعل دفاعي – ونضيف فحوص nil واضحة لكل طريقة، بحيث يهلع البرنامج لو استدعاء المستخدم طريقة بعد Close عن طريق الخطأ.

إن writer ليست آمنة تزامنيًا، وليس هذا وحسب، بل أن الاستدعاءات المتزامنة لـ Close و Write ستجعل البرنامج ينهار في شفرة C. إن إصلاح هذا هو التمرين رقم 13.3.

إن برنامج bzipper أدناه هو أمر ضاغط bzip2 الذي يستخدم حزمتنا الجديدة. وهو يعمل كأمر bzip2 الموجود في العديد من نُظم Unix.

```go
gopl.io/ch13/bzipper

// Bzipper reads input, bzip2-compresses it, and writes it out.
package main
import (
    "io"
    "log"
    "os"
    "gopl.io/ch13/bzip"
)
func main() {
    w := bzip.NewWriter(os.Stdout)
    if _, err := io.Copy(w, os.Stdin); err != nil {
        log.Fatalf("bzipper: %v\n", err)
    }
    if err := w.Close(); err != nil {
        log.Fatalf("bzipper: close: %v\n", err)
    }
}
```

سنستخدم bzipper في الجلسة أدناه لضغط /usr/share/dict/words، وهو قاموس النظام، من 938848 بايت إلى 335405 بايت – أي حوالي ثلث حجمه الأصلي – ثم نضغطه بأمر bunzip2 الخاص بالنظام. إن تلبيد (hash) SHA256 يظل كما هو قبل وبعد، ويمنحنا ثقة في أن الضاغط يعمل بطريقة صحيحة. (لو لم يكن لديك sha256sum في نظامك، استخدم حلّك للتمرين 4.2).

```
$ go build gopl.io/ch13/bzipper
$ wc -c < /usr/share/dict/words 938848
$ sha256sum < /usr/share/dict/words
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
$ ./bzipper < /usr/share/dict/words | wc -c
335405
$ ./bzipper < /usr/share/dict/words | bunzip2 | sha256sum
126a4ef38493313edc50b86f90dfdaf7c59ec6c948451eac228f2f3a8ab1a6ed -
```

لقد أوضحنا كيفية ربط مكتبة C مع برنامج Go. ولو نظرنا للأمر من الاتجاه الآخر، سنجد أنه من السهل ترجمة برنامج Go كأرشيف ثابت يمكن ربطه ببرنامج C أو كمكتبة مشتركة يمكن تحميلها ديناميكيًا بواسطة برنامج C. لقد تطرقنا للأجزاء السطحية البسيطة فقط من cgo هنا، وهناك الكثير جدًا من الأشياء الأخرى المتعلقة بإدارة الذاكرة، والمؤشرات، والاستدعاءات، والتعامل مع الإشارة، والسلاسل والأخطاء وأدوات الإنهاء، والعلاقة بين goroutines وتشعبات نظام التشغيل، وجزء كبير من كل هذا هو أمور دقيقة تحتاج لبراعة في التعامل. إن قواعد تمرير المؤشرات من go إلى C والعكس بطريقة صحيحة معقدة بشكل خاص، لأسباب مشابهة لتلك التي ناقشناها في القسم 13.2، ولم تُحدد بشكل رسمي بعد. لو أردت معرفة المزيد من المعلومات عن الأمر، ابدأ بـ : https://golang.org/cmd/cgo

تمرين 13.3: استخدم sync.Mutex لجعل bzip2.writer آمن للاستخدام المتزامن بواسطة goroutines متعددة.

تمرين 13.4: الاعتماد على مكتبات C له عيوبه. قدم تطبيق بديل بلغة Go خالصة لـ bzip.NewWriter يستخدم os/exec لتشغيل /bin/bzip2 كعملية فرعية.

13.5 تنبيه آخر

ختمنا الفصل السابق بتنبيه حول عيوب واجهة الانعكاس. ينطبق هذا التنبيه بقوة أكثر على حزمة unsafe التي قدمناها في هذا الفصل.

تعزل اللغة عالية المستوى البرامج والمبرمجين عن المواصفات المبهمة في مجموعات تعليمات الكمبيوتر الفردية، وليس هذا وحسب، ولكنها تعزلهم كذلك عن الاعتماد على أشياء غير ذات صلة مثل مكان وجود المتغير في الذاكرة، وحجم نوع البيانات، وتفاصيل مخطط البنية، ومستضيف تفاصيل التطبيق الأخرى. ونتيجة هذه الطبقة العازلة، يصبح من الممكن كتابة برامج آمنة وقوية تعمل على أي برنامج تشغيل دون تغيير.

تسمح حزمة unsafe للمبرمجين بتجاوز طبقة العزل لاستخدام خاصية حَرِجة ولكن لا يمكن الوصول لها إلا بهذه الطريقة، أو ربما يتجاوزونها من أجل تحقيق أداء أعلى. يأتي هذا عادة على حساب القابلية للنقل والأمان، وبالتالي يستخدم الفرد unsafe على مسئوليته الخاصة. إن نصيحتنا حول كيف ومتى يتم استخدام unsafe تشبه تعليقات Knuth التي قدمناها في القسم 11.5 حول التحسين السابق لأوانه. لن يحتاج معظم المبرمجين لاستخدام unsafe على الإطلاق. بالرغم من هذا، قد تنشأ مواقف من حين لآخر سيكون من الأفضل فيها كتابة جزء حَرِج من الكود باستخدام unsafe. لو أوضحت الدراسة والقياسات الدقيقة أن unsafe هي أفضل طريقة ممكنة فعلاً، فاجعل الأمر مقصورًا على منطقة صغيرة قدر الإمكان، بحيث لا ينتبه معظم البرنامج لاستخدامها.

اخرِج آخر فصلين من ذهنك مؤقتًا، واكتب بعض برامج Go المهمة. تجنب reflect و unsafe، وعُد لهذه الفصول عند الضرورة فقط.

نتمنى لك برمجة Go سعيدة، ونأمل أن تكون قد استمتعت بالكتابة بـ Go كما استمتعنا نحن.